U.S.N.A.—Trident Scholar project report; no. 369 (2008)

COMPARING THROUGHPUT AND POWER CONSUMPTION IN BOTH SEQUENTIAL AND RECONFIGURABLE PROCESSORS

by

Midshipman 1/c Kevin K. Liu
United States Naval Academy
Annapolis, Maryland

_____

(signature)

Certification of Advisers Approval

CDR Charles B. Cameron, USN
Department of Electrical and Computer Engineering

_____

(signature)

_____

(date)

Professor Antal Sarkady
Department of Electrical and Computer Engineering

_____

(signature)

_____

(date)

Acceptance for the Trident Scholar Committee

Professor Joyce E. Shade
Deputy Director of Research & Scholarship

_____

(signature)

_____

(date)

USNA-1531-2

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATE COVERED
	5 May 2008	

4. TITLE AND SUBTITLE
Comparing Throughput and Power Consumption in Sequential and Reconfigurable Processors

5. FUNDING NUMBERS

6. AUTHOR(S)

Liu, Kevin K.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)		8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
US Naval Academy Annapolis, MD 21402		Trident Scholar project report no. 369 (2008)

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
This document has been approved for public release; its distribution is UNLIMITED.	

13. ABSTRACT This research project involves an investigation of parallel processing using reconfigurable logic devices. The goal of this project is to support the Naval Research Labs' recent acquisition of a Cray XD-1 supercomputer. A feature of the Cray XD-1 is that it contains field programmable gate arrays (FPGAs). These reconfigurable devices contain hardware whose connections can be modified to target a specific computation. This adaptability can significantly improve the processing speed of computationally intensive operations. Recent improvements in the memory capacity of FPGAs have spurred interest in using the devices for arithmetic floating-point operations using the IEEE 754 standard. However, adapting a program designed to run on a sequential processor to be run instead on an FPGA can be time consuming and difficult for anyone lacking significant experience in hardware design. In this project, a high-level language (HLL)—Mitrion-C 1.4—was used to reduce some of this effort. Using this language, two calculations taken from a ray-tracing simulation of NASA's Moderate Resolution Imaging Spectroradiometer (MODIS) were implemented on an FPGA. The calculations consisted of floating-point additions, subtractions, multiplications, divisions, and square root extractions. It was feasible to perform many of the calculations in parallel, leading to a substantial increase in system throughput. Functionally identical programs were also implemented on a sequential processor—an Opteron 275—using the American National Standards Institute's standard for C (ANSI-C). Those portions of the FPGA design and of the sequential programs that were dedicated to performing scientific calculations were isolated and their processing time was measured using functions written in ANSI-C and calculated by the sequential processor. In addition, power consumption was measured both while the FPGA hardware implementation ran and while the sequential program ran. The results showed that implementing the two calculations on an FPGA was about 900% faster than a sequential processor, requiring only roughly a 30% increase in power consumed.

14. SUBJECT TERMS	15. NUMBER OF PAGES
field programmable gate arrays, floating point arithmetic, high-performance reconfigurable computing, Mitrion-C, power consumption	**83**
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Abstract

This research project involves an investigation of parallel processing using reconfigurable logic devices. The goal of this project is to support the Naval Research Labs' recent acquisition of a Cray XD-1 supercomputer. A feature of the Cray XD-1 is that it contains field programmable gate arrays (FPGAs). These reconfigurable devices contain hardware whose connections can be modified to target a specific computation. This adaptability can significantly improve the processing speed of computationally intensive operations.

Recent improvements in the memory capacity of FPGAs have spurred interest in using the devices for arithmetic floating-point operations using the IEEE 754 standard. However, adapting a program designed to run on a sequential processor to be run instead on an FPGA can be time consuming and difficult for anyone lacking significant experience in hardware design. In this project, a high-level language (HLL)—Mitrion-C 1.4—was used to reduce some of this effort. Using this language, two calculations taken from a ray-tracing simulation of NASA's Moderate Resolution Imaging Spectroradiometer (MODIS) were implemented on an FPGA. The calculations consisted of floating-point additions, subtractions, multiplications, divisions, and square root extractions. It was feasible to perform many of the calculations in parallel, leading to a substantial increase in system throughput. Functionally identical programs were also implemented on a sequential processor—an Opteron 275—using the American National Standards Institute's standard for C (ANSI-C).

Those portions of the FPGA design and of the sequential programs that were dedicated to performing scientific calculations were isolated and their processing time was measured using functions written in ANSI-C and calculated by the sequential processor. In addition, power consumption was measured both while the FPGA hardware implementation ran and while the sequential program ran. The results showed that implementing the two calculations on an FPGA was about 900% faster than a sequential processor, requiring only roughly a 30% increase in power consumed.

Keywords: field programmable gate arrays, floating point arithmetic, high-performance reconfigurable computing, Mitrion-C, power consumption

Acknowledgements

I would like to thank the United States Naval Academy Electrical Engineering Department for providing their support and equipment for this project. In particular, I would like to thank my advisers CDR Charles B. Cameron and Professor Antal A. Sarkady for their support, patience, and instruction.

This work was supported in part by a grant of computer time from the DOD High Performance Computing Modernization Program (DoD HPCMP) at the Naval Research Laboratory (NRL). I would like to thank the many scientists including Wendell Anderson, Rick Hurd, Jeanie Osburn, and Ray Yee at NRL who willingly shared their time and expertise. This work was also supported by Kenneth Sarkady, Head of the Infrared Countermeasures Systems Section.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Naval Research Laboratory (NRL) acquired a Cray XD1 supercomputer in 2005. The system uses 840 AMD Opteron 275 Dual-Core processors and 144 Xilinx Virtex-II Pro Field-Programmable Gate Arrays (FPGAs) [1]. The system was purchased by the Center for Computational Science, located in the Information Technology Division, and seeks to provide high-performance computing (HPC) resources for Department of Defense (DoD) research [2]. high-performance computing describes the use of processors or computing nodes connected in parallel to perform supercomputing. The performance of an HPC system is typically measured in FLOPS, which refers to Floating-point Operations Per Second.

In traditional HPC a given application is split between large numbers of commercially-available sequential processors running in parallel. This approach permits high throughput at relatively low expense. However, conventional processors are typically designed to be used for sequential operations. When heavily parallel problems are implemented on them, these processors often cannot reach full utilization. Although connecting many nodes in parallel has allowed HPC to be done using conventional processors, portions of each processor needed for other applications might sit idle during a parallel task, thus resulting in inefficiencies in both cost and performance.

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices containing many "logic blocks" that can be reconfigured to perform basic logic functions, such as AND, OR, and NOT. These basic logic functions are the foundations of all computing tasks and any other application, including arithmetic, can be created using only these functions. Because they are reprogrammable, FPGAs have in the past been used to test circuit designs before mass production. However, recent advances in FPGA technology have made it to feasible to perform floating-point operations on FPGAs.

NRL purchased the Cray XD1 to test the applicability of using FPGAs to accelerate Navy and Department of Defense applications. Despite previous research that shows that floating-point operations are not only possible on FPGAs but should be accelerated by their use, the fact remains that customizing an FPGA for a specific application is generally regarded as a time-consuming and technically difficult process. Several techniques have been applied to simplifying the process of programming an FPGA, to varying levels of success [3]. This project gathered data on cost versus benefit when implementing a particular application on FPGAs.

In this project the IEEE 754 standard [4] was used for floating-point representation. Although past research [5], [6], [7], has found that representing floating-point numbers using custom formats on FPGAs typically requires fewer resources from the FPGAs and can be run faster, the scientific community has largely adopted the IEEE 754 standard.

A specific application was selected to investigate floating-point operation performance. The application was an optical simulation of NASA's Moderate Resolution Imaging Spectroradiometer (MODIS). The MODIS simulation was chosen for three reasons. First, previous research has shown that the problem is highly amenable to parallel processing [8]. Second, the problem can use the IEEE 754 standard. Finally, the problem requires implementation of floating-point addition, subtraction, multiplication, division, and square root. Since these mathematical functions are the most used in the vast majority of possible scientific functions, a study into their performance on FPGAs is applicable to FPGA floating-point operations in general.

A cost-versus-benefit analysis comparing current FPGA technology to available conventional processors was sought in this project. Modern HPC systems are expensive to operate because of power and heat requirements. Initial experiences with high-performance computing using FPGAs have suggested that total power consumption can be reduced because FPGAs operate at lower clock speeds and so draw less power per chip. The only way FPGAs can show improvements in throughput over processors with higher clock speeds, therefore, is if they are customized so that their resources are more heavily used.

Although the specifications of current technology indicate that employing FPGAs over conventional processors would be more cost-effective, these assertions have yet to be quantitatively confirmed with a realistic application. The FPGAs available on the Cray XD1 have access to 16 megabytes (MB) of memory for input and output [1]. Many applications require more than 16 MB, so in those cases, a conventional processor with access to larger banks of memory repeatedly transfers data between the FPGA and external memory. The effects on power requirements and throughput of this use of conventional processors in conjunction with FPGAs are not well documented. In this project, a practical scientific application was implemented on commercially available reconfigurable devices in order to generate a cost-benefit comparison for an application using both FPGAs and conventional processors.

Chapter 2

Background

2.1 Field-Programmable Gate Arrays

A useful primer to understanding FPGA architecture is Brown and Rose's *Architecture of FPGAs and CPLDs: A Tutorial* [9]. Field-Programmable Gate Arrays are a type of Field-Programmable Device (FPD). FPDs are sometimes also known as Programmable Logic Devices (PLDs). These terms generally describe devices that can be configured by the user to implement hardware designs. Using FPDs allows designers to test their designs without having to incur the high fixed costs associated with custom-designed integrated circuits.

Traditional semiconductor devices implement hardware designs by creating devices and the electrical connections between them on a single integrated circuit. The precursor to the FPGA was the Mask-Programmable Gate Array (MPGA). These devices consisted of transistors, the basic building blocks of almost all electrical circuitry, in an array that could be connected physically at the time of manufacture to realize a circuit design. However, this technique required that a customized chip be fabricated, an expensive process because of high associated fixed costs. The FPGA applies the concept of an MPGA but is implemented using user-programmable technology.

Figure 2.1 on the following page shows the architecture of an FPGA. Each of the input/output (I/O) blocks can be configured for input, output, or bidirectional behavior. The logic blocks can be configured to behave as any combination of logic gates. The physical connections between logic blocks can be switched on or off by the user to connect logic blocks without the need for physical fabrication [10]. The figure gives an idea of how any I/O or logic block can potentially be connected to any other block by reconfiguring the FPGA's network of connections.

Development of FPDs has recently been focused mostly on FPGAs because they employ Dynamic Random Access Memory (DRAM), and so have a higher logic capacity than other FPDs. Logic capacity refers to the amount of logic that can be mapped to a given FPD. It is usually compared to the equivalent number of logic gates that would be available on a traditional gate array [9].

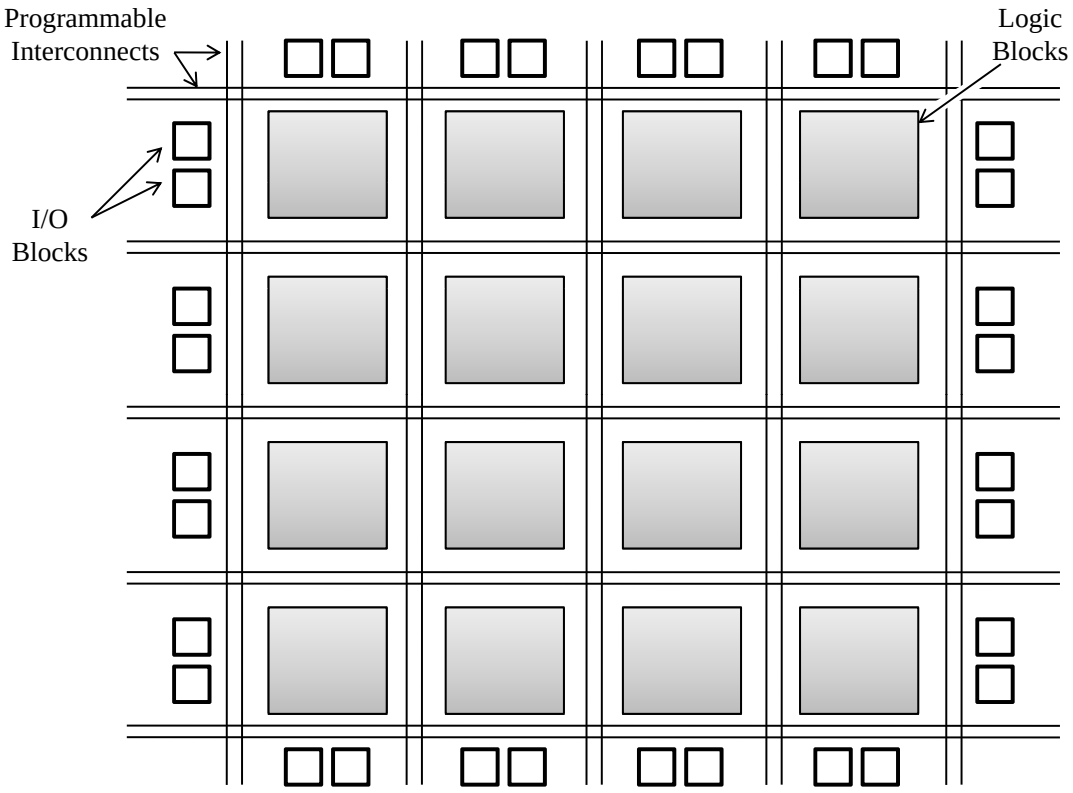

Figure 2.1: FPGA architecture.

2.2 Software Development vs. Hardware Design

In this report, the terms *software development* and *hardware design* are used to distinguish between two different production methodologies. Whereas software engineers are most concerned about the correctness of their algorithms and can ignore many hardware constraints, hardware designers cannot afford to do so: from the outset they must consider the actual physical limitations of the device they are working with. This is also why what could be considered a “program” for an FPGA is known instead as a design—because hardware code is literally mapped to the physical components of a device, and so has more in common with circuit design than software programs.

Working with both hardware and software, as is the case in this project, can present challenges because of overlapping terminology and cultural differences between the two fields. The main difference lies in the level of abstraction. Software is created at a high level of abstraction. This means that software developers work with logic, but the implementation of that logic is taken care of by the target device (a processor). The benefit of high-level programming is that software can usually be run on a wide variety of devices without device-specific customization.

Hardware design functions in the "real world" of physical things. Hardware designers must

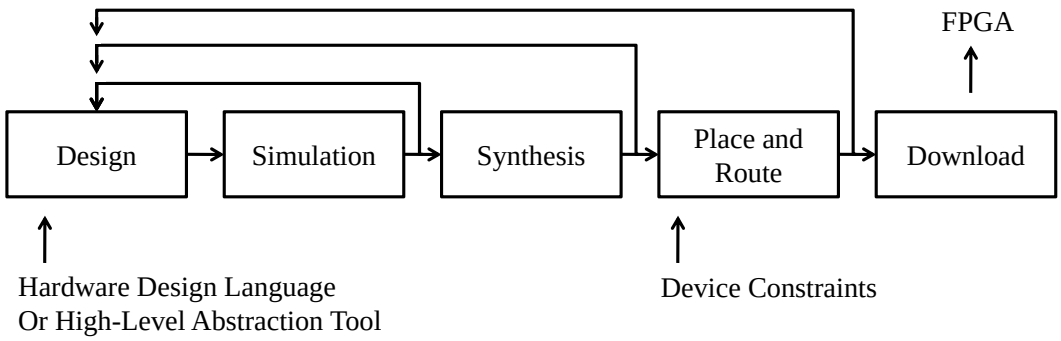

Figure 2.2: Hardware design flow.

be aware of their target device's physical constraints, such as available memory, logic blocks, and physical connections between elements. Working with such a low level of abstraction, that is, at such a high level of detail, has traditionally meant that hardware design takes significantly longer than software development [11].

Efforts have been made to increase the abstraction level available for hardware design so as to allow researchers with less experience in the field to customize hardware for their projects. However, attempts to bridge software and hardware have faced difficulty in overcoming a basic difference in design methodology. Software developers design sequential code. That is, they act as if each line of code is run after every line preceding it. Hardware developers, on the other hand, must always think in parallel. All blocks within a hardware design are synchronized by a clock, but can process inputs and outputs independently of all other blocks. Figure 2.2 shows the hardware design process and is discussed below.

Hardware design can begin at a relatively high level of abstraction if a high-level language (HLL) is used. The development of these tools is further discussed in section 3.1 on page 18. After the design stage, a simulator is used to test that the logic of a hardware design is functionally correct. While this step would essentially be the last step in software development, a hardware design must go through several more processes before it can be loaded onto a device. Compilation of a hardware design requires two steps. First, the synthesis step generates a device-independent intermediate representation of the design. Synthesis tells the designer how many resources a design will require; if the target hardware lacks sufficient resources, the designer must return to the design stage. Place-and-route is the second compilation step and is only run if synthesis completes successfully.

In the place-and-route step, the structures generated in synthesis are mapped to physical components on the FPGA. This process is device dependent and will fail if the required resources are not available on the target device. Place-and-route is the most time consuming step of hardware design, taking from thirty minutes to many hours depending on the complexity of the design and the hardware specified. The output of this step is called a bitstream, a mapping of binary values to specific blocks and connections on a device. If place-and-route or synthesis fails, the designer

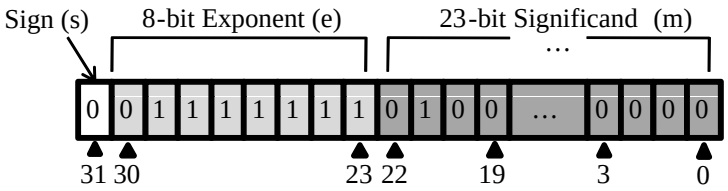

Figure 2.3: IEEE 754 Single-precision floating-point representation.

must return to the design step to adjust the design, as shown in Figure 2.2 on the previous page. The final step of the process is the download of the bitstream onto hardware. The specifics of this step depend on the particular hardware being used [10].

2.3 IEEE 754 Single-Precision Floating-Point Representation

Floating-point representation is a system used for representing real numbers. The name derives from the fact that the location of the decimal point or radix point of the number being represented is variable. A floating-point number is often known as a *float*. The floating-point format differs from the fixed-point number representation system, in which the location of the decimal point is constant. For example, integer representations of numbers are a fixed-point representation because all numbers have exactly zero decimal places. The benefit of using floating-point representation is its ability to represent a greater range of values, which is often important to scientific applications. The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the industry standard for the representation of floating-point values today [4]. It allows numbers to be represented as binary strings of 1's and 0's, which is important for computing applications. There are four floating-point representation formats defined by IEEE 754, of which only two are commonly used: single-precision and double-precision. The IEEE 754 standard only requires that 32-bits be used in the single-precision representation—other bits are optional. Single-precision numbers are adequate for many scientific applications and the double-precision standard takes significantly more resources to implement on FPGAs. Therefore, only the single-precision floating-point format was considered in this project.

As mentioned before, the IEEE 754 single-precision format represents a real number using a string of 1's and 0's. Figure 2.3 shows how the 32 bits of a single-precision number are broken down. In the figure, a given floating-point number f can be represented as the equation

$$f = (-1)^s \times 2^{(e-127)} \times m \tag{2.1a}$$

In this equation, $s = 1$ when bit $31 = 1$ and $s = 0$ when bit $31 = 0$. The exponent field e is adjusted by 127 to allow the exponent to range between -126 and 127. The values of e that would represent exponents of -127 and 128 are instead used for special cases, as described later in this section. The significand, m, always has a leading bit of 1 for normalized numbers, so only the

Table 2.1: Truth table comparing XOR and binary addition.

X	Y	X XOR Y	$X+Y$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	(1)0

fractional part is stored in the floating-point format. By definition, $1 \leq$ significand < 2. Consider the example of the floating-point representation of the decimal value 5.0 . The number would be represented as the binary string $0\,10000001\,01000000000000000000000_2$ using the IEEE 754 format. The description of the IEEE format, above, explains how the 32-bit binary string represents a decimal number. The sign bit s is 0, so the number is positive. The exponent field e is 10000001_2, which converts into $1 \times 2^7 + 1 \times 2^0 = 128 + 1 = 129$. This means that the actual exponent is $129 - 127 = 2$. Finally, the significand is $(1.)01000000000000000000000_2$, or $1 + 1 \times 2^{-2} = 1.25$. After calculating these values, f can be solved $f = 1 \times 2^2 \times 1.25 = 5.0$.

In addition to representing real numbers, the IEEE 754 standard also provides codes for infinity and not-a-number, abbreviated NaN. These codes result from underflow or overflow, which occur when f exceeds the range of the IEEE 754 standard, or from division by 0. In addition, *denormalized* numbers are numbers where both the exponent e and the leading bit of the significand are 0. This format allows representation of the numbers in the range $-1^s \times 0.frac \times 2^{-126}$ where *frac* is the fractional part of the significand m. This representation only uses a portion of the precision of the significand. These special cases of the IEEE 754 standard are necessary for many applications. They are also significant to the implementation of floating-point operations on hardware because any system that uses the IEEE 754 standard must devote logic to dealing with these special cases [4].

2.4 Mathematical Operations with Floating-point Numbers

Floating-point arithmetic is considerably more taxing on computer systems than fixed-point arithmetic. The additional cost in resources is not because of increased memory requirements—it is much more difficult to multiply two 32-bit IEEE 754 floating-point numbers than it is to multiply two 32-bit fixed-point numbers. The difficulty of performing floating-point arithmetic comes from the complexity of its algorithms. This section contrasts the logic behind fixed-point and floating-point multiplication as an example of this complexity.

Multiplying two fixed-point binary integers with digital logic is relatively simple. Multiplication is repeated addition. For binary integers, addition follows almost the same rules as the eXclusive OR (XOR) logic gate, as shown in table 2.1.

XOR returns 1 whenever exactly one of its inputs is 1. Line 4 of the table shows that when both X and Y are 1, the sum is 0 with a carry bit of 1, depicted as (1). Resolution of the carry bit requires

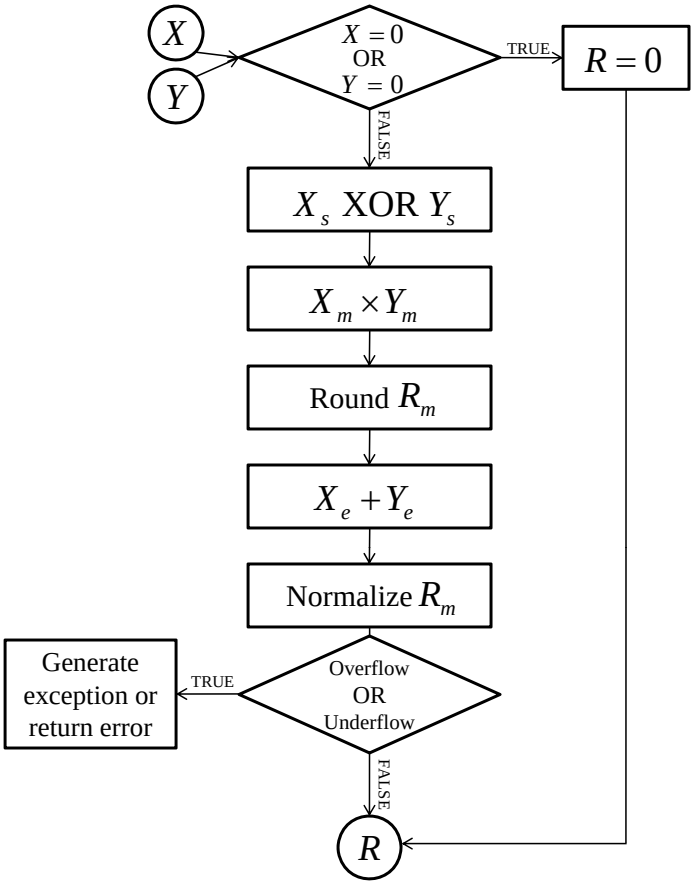

Figure 2.4: Floating-point multiplication algorithm.

additional logic. Even so, the gates required overall for binary addition are common in digital logic and require few resources to implement. Fixed-point multiplication can be implemented by using this algorithm repeatedly. Multiplication of floating-point numbers, on the other hand, requires several stages and different types of logic. Figure 2.4 shows the required logic flow and is discussed in the paragraphs below.

In the figure, the subscript s refers to the sign of a number, e to the exponent, and m to the significand. The two inputs are X and Y and the output is R. The algorithm first checks whether either X or Y is zero, in which case R is set to zero and the multiplication ends. Otherwise, the resultant sign of R is computed by comparing the sign bits of X and Y using an XOR gate. Next, the fractional parts of the significands of the inputs are multiplied, using a fixed-point multiplication method. The result (R_m) must be rounded to fit within the single-precision standard. To calculate the exponent of the result (R_e) the exponents of the inputs are added. Also, if R_m is not between 1 and 2, as the IEEE format requires, it must be normalized. This requires a shift operation to be performed on R_m and for R_e to be incremented or decremented, as appropriate. Finally, the result is checked for overflow and underflow, which would change the result to either infinity or

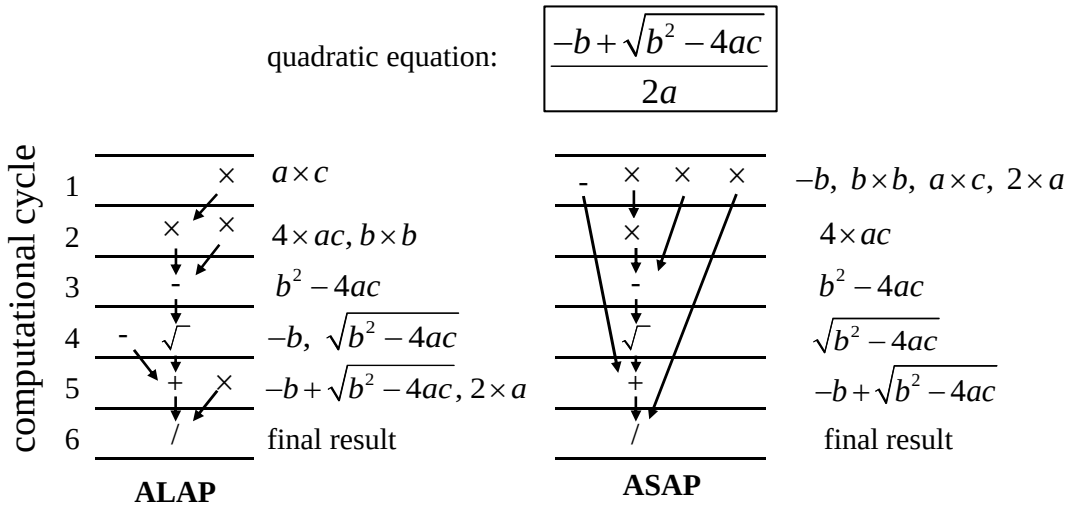

Figure 2.5: Two scheduling techniques applied to a single task.

NaN. The example given demonstrates the potential complexities hardware designers face when implementing floating-point arithmetic on reconfigurable logic.

2.5 Scheduling

Many techniques for implementing floating-point operations on FPGAs exist. Customizations to data storage and allocation of resources have been made to increase floating-point performance. In general, one customization important to any supercomputing application is scheduling. Scheduling refers to the order, or priority, given to tasks in a multi-task system. There are many different scheduling techniques, each offering unique benefits, and scheduling of resources in an FPGA is especially important for fast performance. This section summarizes some basic scheduling techniques relevant to this project.

Within the broad topic of scheduling, pipelining describes the processing of multiple stages of the same operation simultaneously. As this is a relatively new field, much of the related terminology is non-standard. The terminology used in this section is adopted from Hsu and Jeang's discussion of pipelining techniques [12] with some modifications made to reflect current common usage. Figure 2.5 shows an example using the quadratic equation. It is an example of what will be referred to as a task. Anything above the task level might be referred to as a system, application, or problem.

In Figure 2.5 on the previous page, a task that computes the quadratic equation is split into subtasks. In this simplified example, each subtask, such as the calculation $a \times c$, requires only one computational cycle to complete. Typically, subtasks such as mathematical operations using floating-point numbers take multiple cycles to complete. The two most widely used scheduling techniques are As-Soon-As-Possible (ASAP) scheduling and As-Late-As-Possible (ALAP) scheduling. As Figure 2.5 on the preceding page illustrates, in ASAP scheduling, subtasks are scheduled as soon as all the subtasks they are dependent upon have been scheduled. In contrast, in ALAP scheduling, the schedule is created from last subtask to first. A subtask is scheduled as soon as all subtasks dependent upon it have been scheduled [12]. ALAP and ASAP scheduling are examples of preprocessing scheduling techniques because they only ensure that no conflicts in dependencies will arise: they do not address the optimization of resource allocation or throughput. For example, the ALAP schedule of Figure 2.5 on the previous page requires at least two multipliers be implemented because two multiplications are scheduled for simultaneous execution during computational cycle 2. In contrast, the ASAP schedule requires at least three multipliers be implemented because three multiplications are scheduled during computational cycle 1. However, both schedules require six total computational cycles to output an answer. Therefore, the ASAP schedule is less efficient in this particular example because it requires more resources to produce the same rate of throughput.

The modulo scheduling technique described by Rau and Glaser [13] measures the effectiveness of a schedule by considering its throughput and the resources required to implement it. Modulo scheduling is distinct from both ASAP and ALAP scheduling. It allows the calculation of the minimum initiation interval, i, that needs to separate the initiation of consecutive tasks. The minimum initiation interval can be calculated as $i = m \times \tau$ where m is the modulus and τ is the clock period. In the example presented in Figure 2.6 on the following page, the computational cycles are without units, so τ simplifies to 1. Ordinarily the modulus of a task is equal to the highest number of operations any one functional unit must perform. In the quadratic example presented earlier, the modulus is 4 because four multiplications are required to generate a solution. No other operation is required more times than this in a single task.

A modulus of 4 means that the minimum initiation interval, i, of the task also equals 4, so one task's solution would be generated every four computational cycles. However, the modulus of a task can be decreased by increasing the number of available functional units. If, for example, two multipliers were implemented in the given example, each multiplier would only have to complete two operations, so the modulus would decrease to 2. Figure 2.6 on the next page shows how an $m = 2$ schedule never requires the use of more than two multipliers simultaneously. This would also allow the minimum initiation to decrease to two computational cycles. If, however, it was necessary to generate one solution every cycle, more multipliers would have to be implemented. In the $m = 1$ schedule four multipliers are required starting at cycle 5. Implementing four multipliers corresponds with a modulus of 1. Similarly, two subtraction units would be needed at this point. The modulus could not descend below 2 unless a second such unit was implemented.

The two schedules of Figure 2.6 on the following page show the considerations that must be taken into account when creating a pipelined schedule of a design. The $m = 1$ schedule would allow the generation of one result every computational cycle. This would be a two-fold throughput

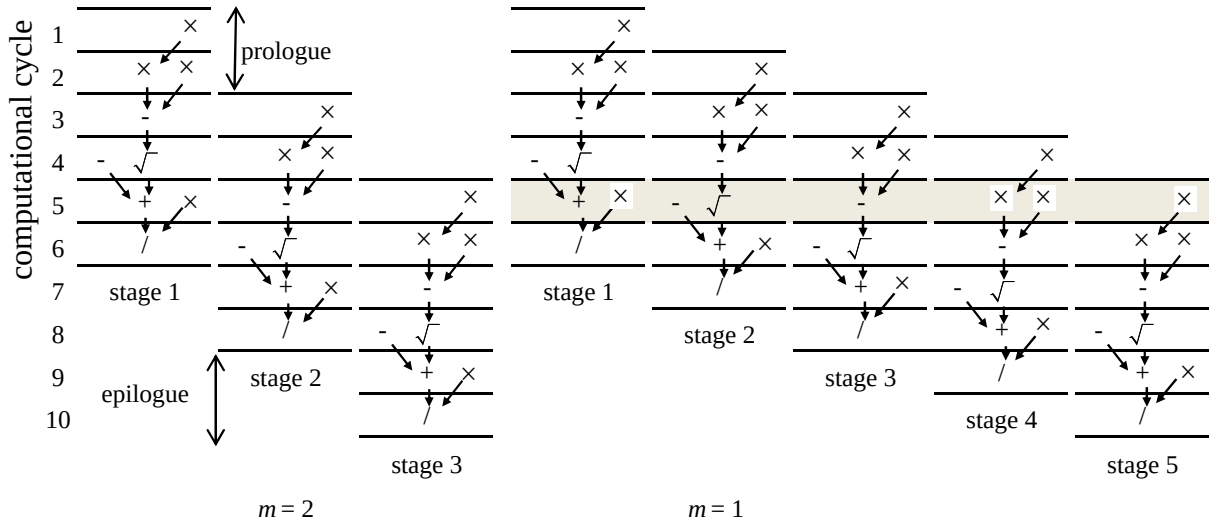

Figure 2.6: Modulo scheduling example

increase over the $m = 2$ schedule, which can only generate one result every two cycles. However, implementing the $m = 1$ schedule would require three additional multiplier units and one additional subtraction unit. This tradeoff between throughput and resource consumption is always on the mind of a hardware designer, since floating-point operations can easily require more resources than available, even in modern FPGAs.

Chapter 3

Related Work

3.1 Implementation of Floating-Point Operations on FPGAs

In 1994, early attempts to implement floating-point operations on FPGAs focused on implementing a design capable of adding and multiplying two IEEE 754 single-precision floats. However, it was discovered that the best design based on a comparison of space versus throughput required more space than was available on a single device. As a result, the design was implemented across four FPGAs [14]. The research's conclusion was that FPGA technology needed to be improved before floating-point could become feasible.

In 1996, implementations of an adder and a multiplier were made to fit onto a single device each. However, the implementations suffered poor performance and accuracy compared to conventional implementations because of resource constraints [15]. Two years later, an IEEE 754-compliant floating-point adder and multiplier were separately implemented on a single FPGA each. It was speculated at this point that floating-point operations on FPGAs could potentially outperform conventional microprocessors in specific circumstances.

The space required to fully implement IEEE 754 floating-point units continued to represent a bottleneck to development, despite the fact that hardware engineers operate at very low levels of abstraction with FPGAs. As a result, several attempts were made to represent floating-point numbers and implement arithmetic operations without using the IEEE 754 standard. Some techniques proved more effective than others. Floating-point to fixed-point conversion involves multiplying a floating-point value by a large number and treating the result as an integer, performing integer arithmetic, and then converting the resulting integer back into a float. This method was shown to be slower and more resource-consuming than a comparable floating-point implementation [16]. By contrast, bit-width optimization, which allows the required accuracy to determine how many bits are actually used to represent a float and only use that many bits, showed improvement over the IEEE 754 implementation [17].

However, techniques that avoided using IEEE 754 were unable to match the standard's precision, range, and treatment of non-real or out-of-bounds situations. Therefore, the industry has for the most part adopted IEEE 754 as standard. The initial problems with implementing IEEE 754 seemed to result from technological rather than methodological limitations. Underwood

made predictions of FPGA IEEE 754 floating-point performance, concluding that FPGAs would show an order-of-magnitude performance advantage over comparable conventional processors by 2009 [18].

These predictions have led the industry to pursue technological development in high-performance reconfigurable computing (HPRC), a term used to describe traditional HPC using FPGAs. At the same time, techniques for automating the FPGA customization process began to be explored. The attempts to extract peak performance from FPGAs described in this section all involved significant knowledge of FPGA architecture and hardware design. Up to this point, designers have used hardware description languages (HDLs) such as Verilog and Very High-Speed Integrated Circuit HDL (VHDL) to describe their algorithms. This code is then compiled by an electronic design automation tool (EDA) into a physical design targeted to a specific device. However, the amount of logic that can be placed on a single chip has grown to the point that very complex algorithms can be implemented on a single FPGA, and so a need for a higher level of abstraction has developed [19].

Developing algorithms at a higher level of abstraction for hardware design also allows software engineers to use FPGAs to accelerate their applications without having to learn an entirely new design methodology. However, the field of HPRC is still in the developmental stages, and high-level languages in particular have a long way to go. A wide range of commercial and open-source HLLs has been developed. Some are easier to use than others, but none has become standard across the industry [3].

3.2 Implementation of the MODIS System

This project applied FPGA floating-point acceleration to a real-world scientific application. This technique was chosen for two reasons. First, it was the intent of this project to be useful to other researchers interested in using FPGAs for software acceleration, but having little background in HPRC, rather than to hardware designers already familiar with the field. Second, a substantial amount of research into HPC and HPRC implementations of the chosen application has already been done, and a cache of data is available for comparison purposes.

The application was an optical ray-tracing simulation of NASA's Moderate Resolution Imaging Spectroradiometer (MODIS). In the spectroradiometer, light from the sun passes through multiple optical elements before reaching a detector. To simulate this system, each interaction of a ray of light with an optical element must be simulated. This interaction entails several important calculations, of which two were of primary focus in this project: (1) finding the point at which a ray intersects an optical surface and (2) finding the direction of the ray's travel after interacting with that surface [20]. These steps can be simplified into a system of floating-point equations with a constant number of inputs and outputs.

The MODIS simulation was initially implemented using Fortran on a Digital Equipment Corporation (since bought by Hewlett-Packard) Alpha 3000 series model 800 computer. The slow performance of this first program prompted interest in methods to speed up processing. Cameron et al. began work in 2002 on implementing the MODIS simulation on multiple digital-signal-processing chips. A system functionally comparable to the original Fortran model was written in

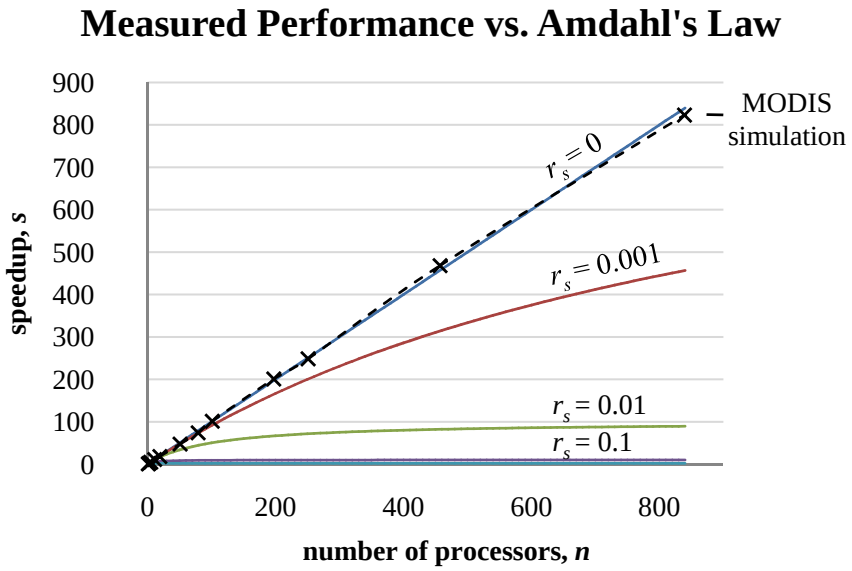

Figure 3.1: Amdahl's law versus measured performance.

the C programming language. The complete simulation was implemented and tested successfully. Initial estimates were that using digital signal processors (DSPs) would show a linear relationship between the number of DSPs used and speedup [21]. This result was confirmed using eight DSPs simultaneously running a complete MODIS simulation [22].

Using the Message Passing Interface (MPI), the MODIS simulation was subsequently implemented on the Naval Research Laboratory's massively parallel Cray XD1. Turn-around-time data was collected for using a single processor and for using multiple processors in parallel [8]. Data points were measured for varying numbers of processors in the range from 1 to 839. These data are reproduced in Figure 3.1 as the line labeled "MODIS simulation".

The performance of parallel computing applications is often measured in speedup, which is the ratio between original throughput and improved throughput, or $s = \rho/\rho_{\text{reference}}$. Figure 3.1 graphs Amdahl's Law, which shows that speedup s can be increased by implementing a parallel application using additional processors. In this specific case, $s = \rho_n/\rho_1$ where ρ_1 refers to the throughput achieved using one processor and ρ_n refers to the throughput achieved using n processors. It also shows that speedup is only maximized when $r_s = 0$, where r_s is the portion of a calculation that must be performed sequentially [23]. The speedup s can be measured using throughput, but s can also be calculated using the formula:

$$s = \frac{1}{r_s + \frac{r_p}{n}} \tag{3.1a}$$

where s is speedup, r_s is the serial component of the process or calculation being implemented, r_p is the parallel component, and n is the number of processors being used for processing. It is also important to note that by definition, $r_s = 1 - r_p$.

This calculation of s allows a designer to predict whether it would be worthwhile to implement a given application using hardware that benefits parallelism. Using multiple processors, as discussed in section 1 on page 7, is one way to implement HPC. To gauge cost versus benefit, the speedup gained would be compared to the extra time, money, and labor required to implement a given application on multiple processors.

The throughput measured when implementing the MODIS system on multiple conventional processors is plotted against examples of Amdahl's Law at different r_s values in Figure 3.1 on the preceding page. Amdahl's Law was used to estimate the components r_s and r_p for the MODIS simulation system. The graph shows that the MODIS simulation had a very low r_s and so an r_p value very close to 1. Therefore, it was concluded that the MODIS simulation would be a good candidate for acceleration using FPGAs, since FPGAs are best suited to highly parallel, data-intensive applications.

The throughput ρ_1 of the MODIS simulation when implemented using a single processor was 6.95×10^6 rays per second. Although the simulation ran very quickly using the Cray's conventional processors, it did not utilize any of the supercomputer's FPGA processing capabilities [8].

Cameron implemented the entire MODIS simulation using the Cray XD1's Advanced Micro Devices Opteron 275 processors. However, such a complex system was not likely to fit on a single FPGA. Instead, only the first two steps of the simulation were implemented on FPGA's in this project. The first step was the calculation of the point where a ray intersected a conicoid surface. The second step was the calculation of the vector normal to the surface at the point of intersection. These steps are referred to as the ray-intersection calculation and the normal-vector calculation, respectively, for the rest of this report. These two calculations are further described in section 4.3 on page 26.

3.3 The Trident Compiler

This project initially selected the Trident compiler for implementation of the normal-vector calculation. The compiler uses a novel approach to convert sequential C code into hardware design language code. It first compiles the C code into an intermediary representation. It then parses this representation to automatically extract parallelisms. Finally, it schedules operations and produces VHDL code based on its analysis of parallelisms.

According to its documentation, the Trident compiler was designed to support ASAP, ALAP, and modulo scheduling (see section 2.5 on page 15 for a description of these methods) among other scheduling methods [24]. To test these claims, the source code of the compiler was modified to implement floating-point arithmetic using Floating-Point Operator v2.0, packaged with Xilinx Integrated Synthesis Environment (ISE) version 8.1i.

Using this modified version of Trident, a simple single-precision floating-point multiplier was implemented and functional accuracy was identified with certainty. However, the scheduling technique being implemented could not be confirmed, since only one arithmetic operation was implemented. Subsequently, the normal-vector calculation was implemented using Trident. However, a simulation of the resulting VHDL code showed anomalies in scheduling. The conclusion was that

that modulo scheduling was not implemented.

Without proper scheduling, the VHDL generated by Trident could not have used the resources of the FPGA adequately. Attempts to contact the developers of Trident to address these issues were unsuccessful. As a result, the focus of this project shifted to a different tool—Mitrion-C.

3.4 Previous Use of Mitrion-C

High-level abstraction tools for hardware design serve two purposes: (1) they simplify the expression of large, complex algorithms, and (2) they simplify hardware design for researchers only familiar with software development. Mitrion-C is a high-level language that is part of the Mitrion Integrated Development Environment (IDE). The function of both Mitrion-C and the Mitrion IDE are further discussed in section 4.1 on the next page

Because of its relatively recent development, little work has been implemented using Mitrion-C. All of the significant related work was published in 2007. Koo et al. compared FPGA performance using Mitrion-C to a software implementation using ANSI-C on the Silicon Graphics, Inc. (SGI) Reconfigurable Application Specific Computing (RASC) RC100 platform, using four Virtex-4 LX200 FPGAs. In the case of an MRI brain scan analysis algorithm, overall speedup was 3.6×, but the speedup of the portion implemented using FPGAs was 11.6× [25].

Koo et al. also used Mitrion-C to implement two other algorithms—the first was a floating-point dense matrix-vector multiplication and the second was an algorithm to simulate solvating protein in water. Comparing the implementation on a single FPGA versus implementation on a single 1.5 GHz Itanium 2 sequential processor, maximum speedup for the first algorithm was 21× and for the second was 10×. Speedup was also shown to increase significantly when using multiple FPGAs [26].

Kindratenko et al. measured speedup comparing the performance of two SGI RC100 FPGAs to that of two 1.4 GHz Intel Itanium 2 sequential processors. Mitrion-C was used to generate the FPGA hardware design. The algorithm concerned the calculation of the two-point correlation function, used to analyze the clustering of extragalactic objects [27]. In the best-case scenario, speedup was measured to be 9.5× [28]. In each of the three reports mentioned above, resource consumption on the target FPGAs was reported and varied from case to case. No correlation between resource consumption and speedup was made.

Speedup using FPGAs was verified by several independent projects, but the benefit of using high-level languages over VHDL or other traditional HDL techniques was not addressed in these reports. El-Araby et al. sought to quantify the "comparative productivity" of various high-level abstraction tools compared with traditional HDL design. Mitrion-C ranked poorly according to the metrics of efficiency and ease-of-use and was also only able to achieve about 60% of the throughput of a manually-coded VHDL solution. However, none of the high-level tools proved to be clearly superior in each of the four applications implemented [29].

Most research using Mitrion-C thus far has focused on speedup. No reports have quantified the relationship between speedup and power consumption. However, Mitrionics AB has repeatedly marketed the Mitrion platform as a low-power solution [30].

Chapter 4

Implementation

4.1 The Mitrion Platform

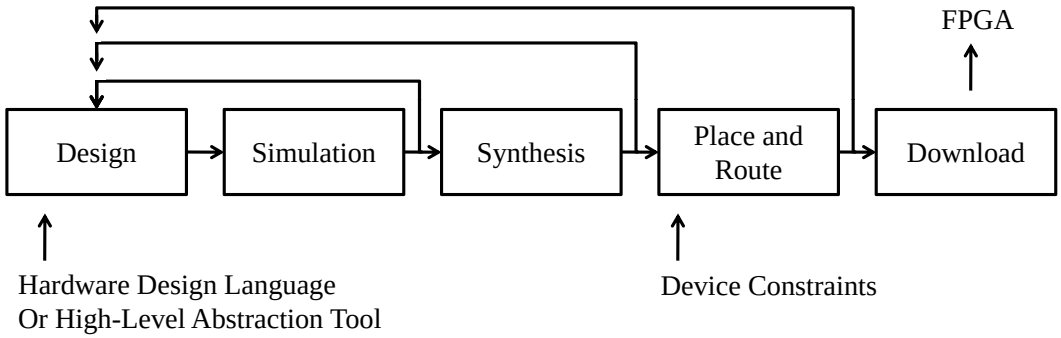

Figure 4.1: Hardware design flow.

The high-level language Mitrion-C is part of the Mitrion Integrated Development Environment (IDE). It is a "C-like" language in that it uses syntax similar to that of the American National Standards Institute's standard for C (ANSI-C). Mitrion-C gives designers the ability to focus on the logic of an algorithm rather than hardware specifics. Parallelism is expressed using data structures and loop constructs. This system gives Mitrion-C the feel of a software language, but allows explicit expression of parallelism as well. The Mitrion IDE converts Mitrion-C programs into VHDL, which can then be synthesized and placed-and-routed using the Xilinx ISE [31].

Generic hardware design flow was first discussed in section 2.2 on page 10. Figure 2.2 on page 11 is reproduced here as figure 4.1 for convenience. Figure 4.2 on the next page illustrates the hardware design flow specific to the implementation using the Mitrion IDE. On the surface, it appears that little difference exists between the two illustrations. In fact, all the same steps are present, since hardware design using Mitrion-C is still hardware design, despite the user interface allowing a higher level of abstraction. However, the Mitrion IDE does offer some benefits over

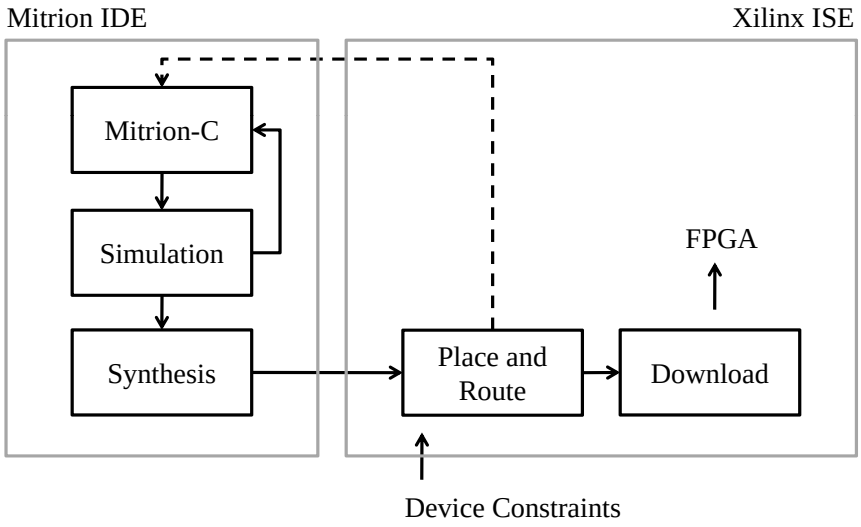

Figure 4.2: Mitrion-C Design Flow.

hardware design using VHDL.

Synthesis of VHDL code can take several minutes to complete and functional simulation is only possible after code has been synthesized. In addition, synthesis will complete successfully even if it is unlikely a design will fit on the target FPGA. In contrast, the Mitrion IDE includes a functional simulator that produces a graphic representation of an algorithm in moments. This simulator will also estimate the amount of resources a design is likely to require.

In sum, the Mitrion IDE allows the user to spend the majority of development time between code development and simulation, as shown by the small loop in figure 4.2. No feedback is needed after synthesis because the Mitrion-C compiler creates the VHDL code automatically. The feedback line after place-and-route is dashed to show that the Mitrion-C compiler will automatically check to make sure a design will most likely fit on the target hardware. This check reduces the number of failures at the place-and-route step significantly.

Mitrion-C and the Mitrion IDE are both part of the Mitrion Software Development Kit (SDK). The Mitrion SDK also contains a graphical simulation tool and libraries that allow a host computer to interface with the Mitrion Virtual Processor (MVP). This processor is the core of the Mitrion Platform. It is a reconfigurable software architecture that runs Mitrion-C code. For each unique application, the Mitrion SDK creates a new virtual processor tailored to the targeted FPGA and optimized for the application [31]. Mitrion-C code defines the customization of each Mitrion Virtual Processor. The use of this intermediary step explains how the user is able to quickly simulate and debug Mitrion-C code: the same virtual processor that can be simulated with the simulator included in the Mitrion SDK can also be loaded directly onto an FPGA.

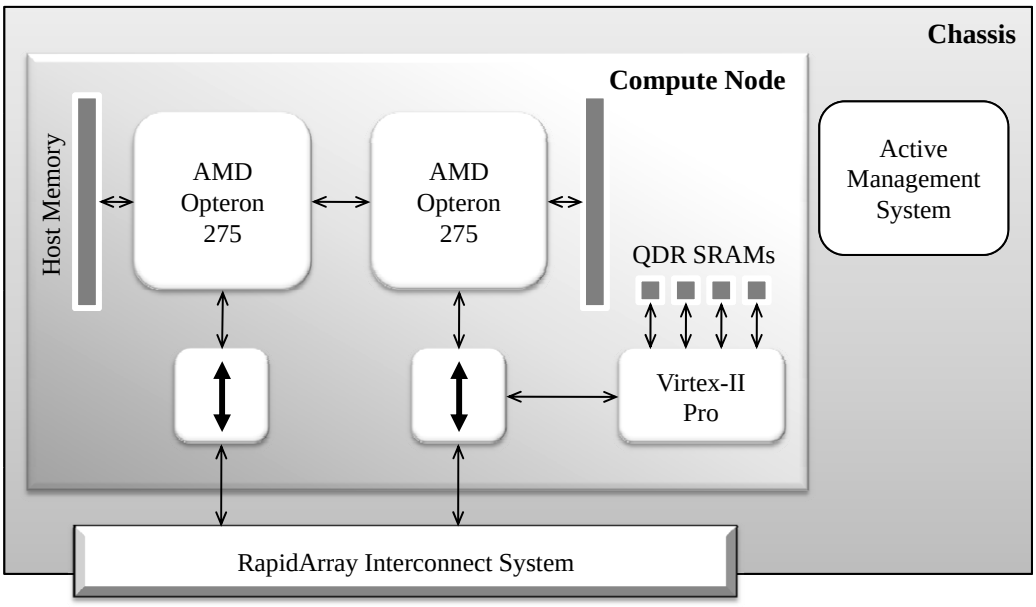

Figure 4.3: Cray XD1 architecture.

4.2 The Cray XD1 Architecture

The Cray XD1 supercomputer is designed to permit high-performance reconfigurable computing (HPRC). Figure 4.3 illustrates the components most significant to this report. Each chassis of a Cray XD1 contains up to 12 AMD Opteron 275 sequential processors, distributed either two or four to each compute node. The research collected in this report used nodes containing two Opterons.

Figure 4.4 on the following page illustrates the memory connections of the Cray XD1. The Opterons have access to host memory, Synchronous Dynamic Random Access Memory (SDRAM) like that found in most personal computers. The Virtex-II Pro FPGAs, however, do not have direct access to this memory. Connected to the FPGAs are four banks of Quad-Data Rate (QDR) Static Random Access Memories (SRAMs), a cache to which the FPGAs have direct access. Each QDR SRAM holds 4MB of memory for a total of cache size of 16MB.

Each memory bus is 64 bits wide, capable of transmitting two 32-bit single-precision floating-point numbers at once. The QDR SRAMs are only accessible to the Opterons through the RapidArray Interconnect System, which is designed to allow communication between FPGA and Opteron with high bandwidth and low latency. The RapidArray Interconnect System also allows the processors of each computing node to communicate with other nodes, but this feature was not used in this project [1].

Although it would seem that access to the QDR SRAMs would require complex C code on the Opterons, Cray created an interface that simplifies the process. The user is given functions that map the Opteron's virtual address space directly into the QDR SRAMs. Therefore, the programmer

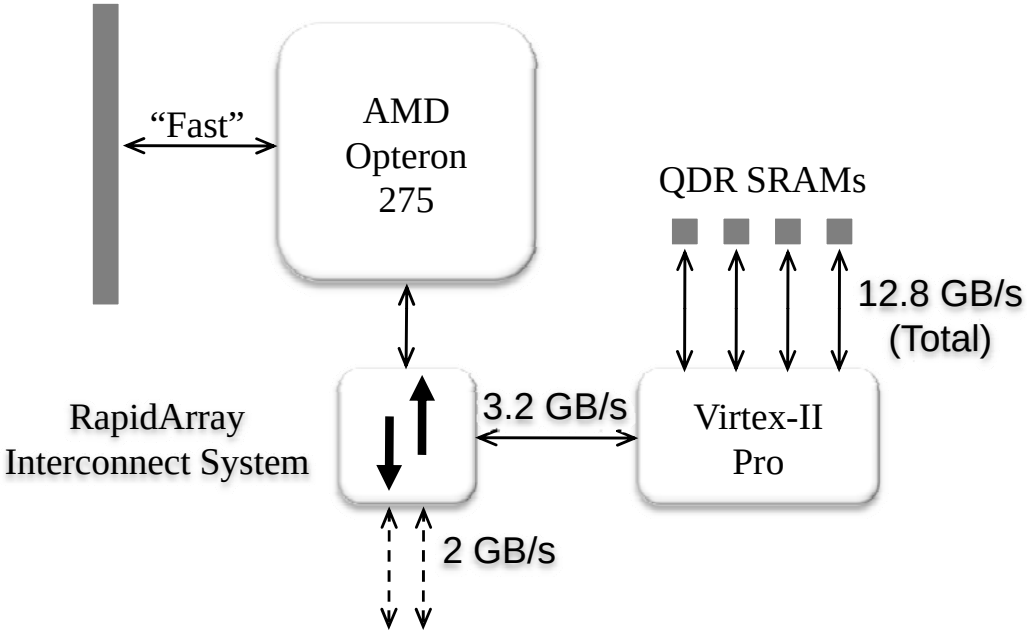

Figure 4.4: Cray XD1 memory connections.

only needs to interact with normal ANSI-C arrays to write to and read from the FPGA's memory. Figure 4.5 on the next page illustrates how the programmer can think of the interaction between Opteron and Virtex-II Pro.

4.3 Description of the Calculations Implemented

As discussed in section 3.2 on page 19, two steps of the MODIS simulation—the ray-intersection calculation and the normal-vector calculation—were implemented in this project. The inputs to the ray-intersection calculation were the point of origin p_0 (given expressed using the Cartesian coordinates x_0, y_0, and z_0), the initial direction vector $\hat{\mathbf{a}}_0$ (given as the direction cosines **L**, **M**, and **N**), the curvature c of the conicoid and the conic constant k, which depends upon the conicoid's type, as shown in table 4.1 on the next page [32].

The output was the point of intersection p_i (given as x_1, y_1, and z_1). Figure 4.6 on page 28 illustrates the calculation. In the figure, the conic surface represents a cross-section of a conicoid. All the points in this cross-section have coordinates with $x = 0$.

The equations with which p_i is calculated are listed in section 4.4.2 on page 32. Once the intersection of a ray with a conic surface has been found, it can be used as the originating point p_0 for the next interaction with an optical element. However, the direction of the resultant ray must first be found. Figure 4.6 on page 28 illustrates the calculation.

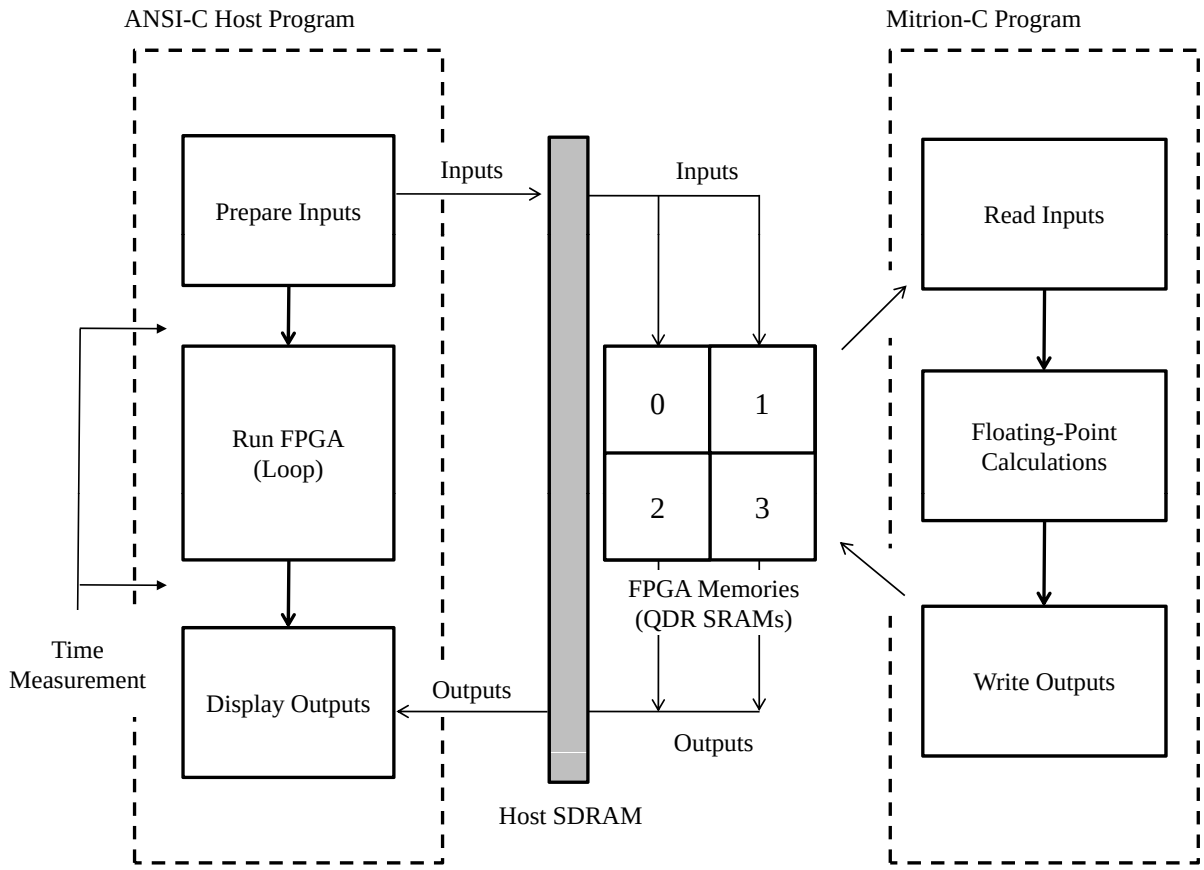

Figure 4.5: Data flow between host and FPGA programs.

Table 4.1: Conic constants and conicoid types.

Conic Constant	Conicoid
$k > 0$	oblate ellipsoid
$k = 0$	sphere
$-1 < k < 0$	prolate ellipsoid
$k = -1$	paraboloid
$k < -1$	hyperboloid

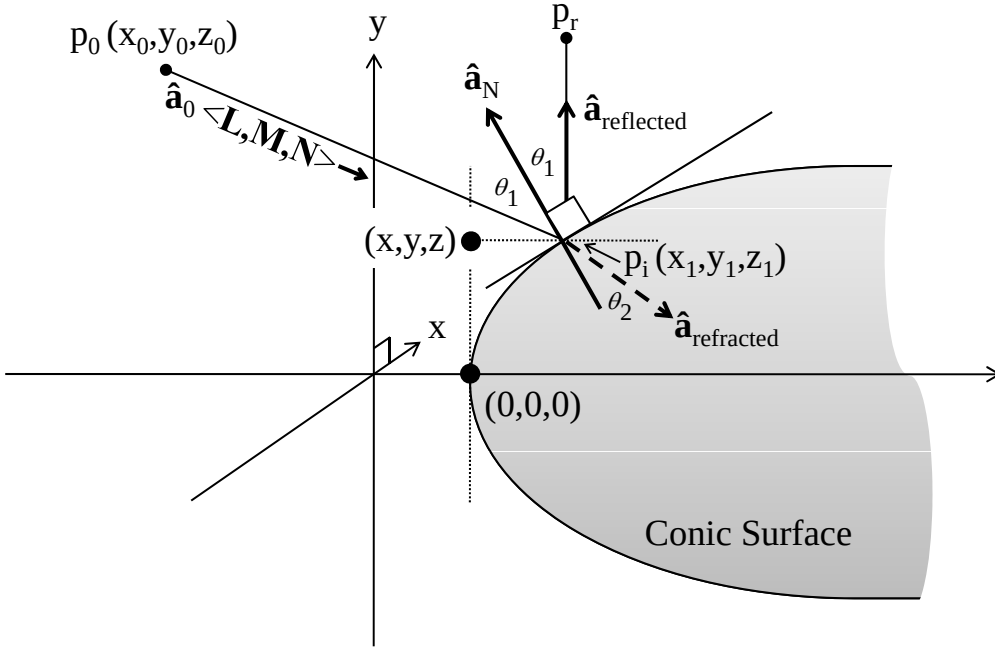

Figure 4.6: Interaction of an incident ray with a conic surface.

After interaction with an optical element, a ray may either be reflected or refracted. The calculation of the ray's path after either reflection or refraction is dependent upon the value of the vector, $\hat{\mathbf{a}}_N$, normal to the conic surface at the point of intersection p_i. In the case of reflection, the angle θ_1 the original ray makes with $\hat{\mathbf{a}}_N$ is equal to the angle between $\hat{\mathbf{a}}_N$ and the unit vector of the resultant ray $\hat{\mathbf{a}}_{\text{reflected}}$.

In the case of refraction, the direction of the resultant ray $\hat{\mathbf{a}}_{\text{refracted}}$ can be found by solving Snell's equation

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \tag{4.1a}$$

or

$$\theta_2 = \arcsin \frac{n_1 \sin\theta_1}{n_2} \tag{4.2a}$$

where both θ_1 and θ_2 are given relative to $\hat{\mathbf{a}}_N$. The calculation of $\hat{\mathbf{a}}_N$ depends on the x and y coordinates at the point of intersection. These coordinates result from the ray-intersection calculation and are provided as inputs to the normal-vector calculation. The approach taken with this problem was to use a different coordinate system with its origin at the vertex of each optical element. Therefore, the origin $(0,0,0)$ represents the vertex of the conicoid.

The normal-vector calculation takes as inputs x_i, y_i, curvature c, and u—a parameter derived from c and the conic constant k. The variable z_i is not needed to calculate the normal vector. The outputs of the calculation are the three components of the resultant normal vector $\hat{\mathbf{a}}_N$ (f, fdx, and fdy). The remaining equations of the normal-vector calculation are listed in section 4.4.1.

4.4 The FPGA Design

4.4.1 Implementation of the Normal-Vector Calculation

The hardware designs loaded onto the Virtex-II Pro FPGAs were created using Mitrion-C. Both the Mitrion-C program for the ray-intersection calculation and the normal-vector calculation used four functions. The normal-vector calculation is discussed first because it is functionally simpler. The source code can be found in appendix B on page 55.

The optical ray-tracing procedure of the normal-vector calculation was explained in section 4.3 on page 26. For the purposes of implementation, the process simplifies into a system of arithmetic operations, represented here:

$$v = u(x^2 + y^2) \tag{4.3a}$$
$$a = \sqrt{1 - v} \tag{4.3b}$$
$$p = 1 + a \tag{4.3c}$$
$$q = ap \tag{4.3d}$$
$$r = pq \tag{4.3e}$$
$$s = 2q \tag{4.3f}$$
$$w = c/r \tag{4.3g}$$
$$b = w(s + v) \tag{4.3h}$$
$$dx = bx \tag{4.3i}$$
$$dy = by \tag{4.3j}$$
$$e = \sqrt{dx^2 + dy^2 + 1} \tag{4.3k}$$
$$f = 1/e \tag{4.3l}$$
$$fdx = fdx \tag{4.3m}$$
$$fdy = fdy \tag{4.3n}$$
$$\hat{\mathbf{a}}_N = (fdx, fdy, f) \tag{4.3o}$$

The problem requires 5 additions, 1 subtraction, 13 multiplications, 2 divisions, and 2 square roots. The addition of constants was implemented as a floating-point operation to ensure precision. The numbers of floating-point units implemented that are reported here reflect the output of the simulator packaged with Mitrion-C.

The *read_inputs()* function reads one 64-bit word (or string of bits) from each of two QDR SRAMs each computational cycle. Since only four inputs were needed for the normal-vector

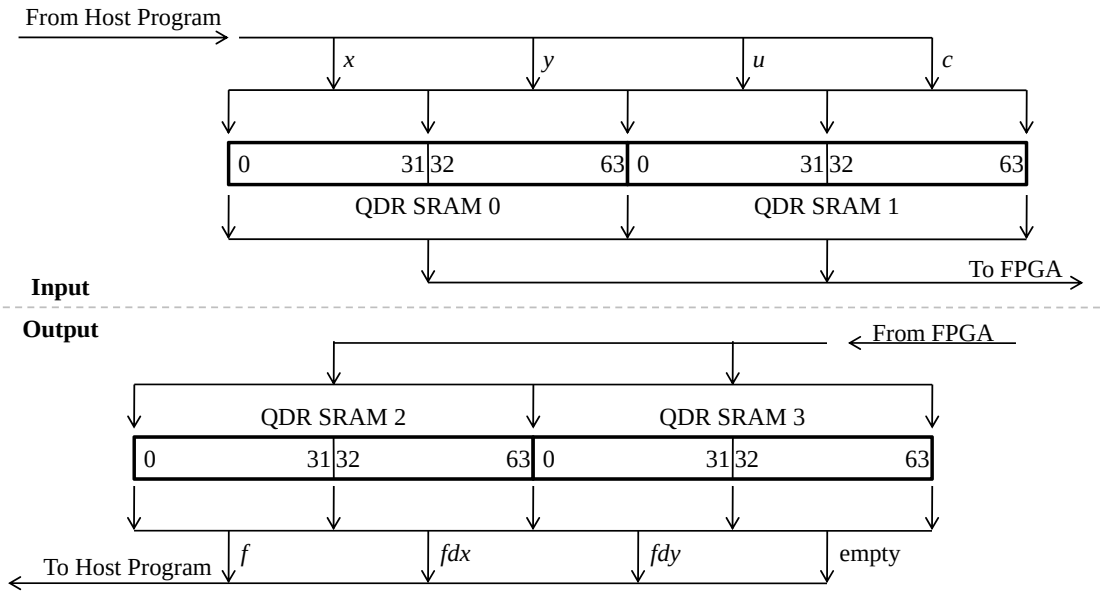

Figure 4.7: Detailed data flow between normal-vector calculation host and FPGA programs.

calculation, QDR SRAMs 0 and 1 were used for input. The reading of inputs from the QDR SRAMs assumes that data is available, that is, that appropriately formatted floating-point inputs have been stored in the memory in the order to be read. This requirement must be satisfied by the host program run on the Opteron. The host program source code for the normal-vector calculation can be found in appendix D on page 67.

The host code associated an ANSI-C array with memory addresses on the QDR SRAMs using the function *mitrion_processor_reg_buffer*. These virtual buffers must be declared as a data type and the buffers' memory addresses (as they appear to the host program) are defined by the size of the data type. Since ANSI-C natively supports 32-bit single-precision floating-point representation as *floats*, the buffers were simply declared as floats.

Data was written by the host program as two 32-bits floats per memory. However, in Mitrion-C the QDR SRAMs may only be read one 64-bit word at a time. The Mitrion-C program was written to read four 32-bit floats as two 64-bit words, split that word into four 32-bit words, and then associate those words with the floating-point format in Mitrion-C. Only then could floating-point arithmetic be correctly implemented on the original four floats. Figure 4.7 illustrates this process, using the normal-vector calculation as an example.

Arithmetic was implemented in the *calc_outputs()* function. Mitrion-C's syntax for arithmetic can be used as if it were sequential ANSI-C. The Mitrion compiler removes the need for the programmer to understand scheduling, floating-point units, or other hardware considerations.

The outputs of *calc_outputs()* were fed to the *write_outputs()* function, which wrote two input floating-point numbers to a QDR SRAM. It first had to pack the floating-point numbers into 64-bit words, reversing the process implemented in *read_inputs()*. Since only three outputs were

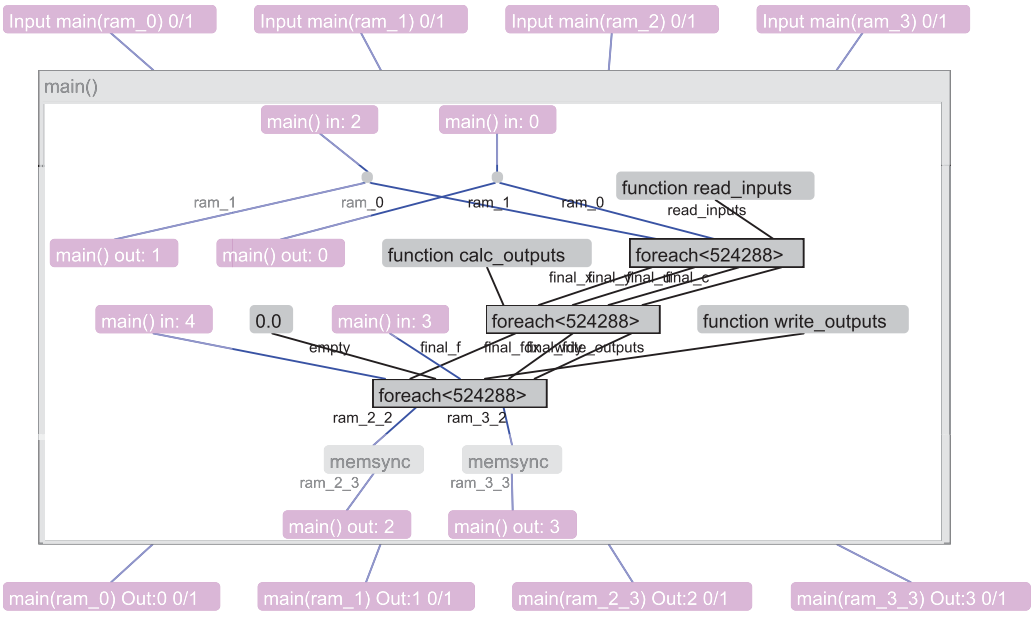

Figure 4.8: Mitrion-C simulation of normal-vector calculation.

generated, the floating-point value 0.0 was written into the second half of QDR SRAM 3 because the 32-bit output could not be written into a 64-bit space. This choice made no difference to throughput since it always takes one computational cycle to perform a memory write, regardless of the value of the actual data.

The three functions *read_inputs()*, *calc_outputs()*, and *write_outputs()* were controlled by the *main()* function. The three functions were implemented in `foreach` loops. In Mitrion-C, when this type of loop is implemented over a *list* of values, the compiler automatically pipelines the code within and executes it in parallel. Further explanation of pipelining and scheduling is provided in section 2.5 on page 15. Figure 4.8 highlights the data dependencies in the normal-vector implementation. Data flows from top to bottom in the figure, but data can be transferred as soon as it is read— that is, the *calc_outputs()* function does not need for every sample to be read by *read_inputs()* before it can begin arithmetic on the inputs.

As mentioned before, the size of each QDR SRAM available to the FPGA was 4 MB. Therefore, the number samples that could be loaded into the memory can be found using:

$$\text{Samples} = 2\,\text{SRAMs} \times \frac{4\,\text{MB}}{\text{SRAM}} \times \frac{1\,048\,576\,\text{bits}}{\text{MB}} \times \frac{1\,\text{float}}{32\,\text{bits}} \times \frac{1\,\text{sample}}{4\,\text{floats}} = 528\,244 = (2^{19}) \quad (4.4a)$$

The FPGA program was looped by the host program 2048 (2^{11}) times, for a total of $2^{11} \times 2^{19} = 1\,073\,741\,824 (2^{30})$ samples, in order to simulate the calculation of a large dataset. The data were generated by the host program and reflected realistic MODIS simulation inputs. Elapsed time was measured using the `clock()` function of ANSI-C.

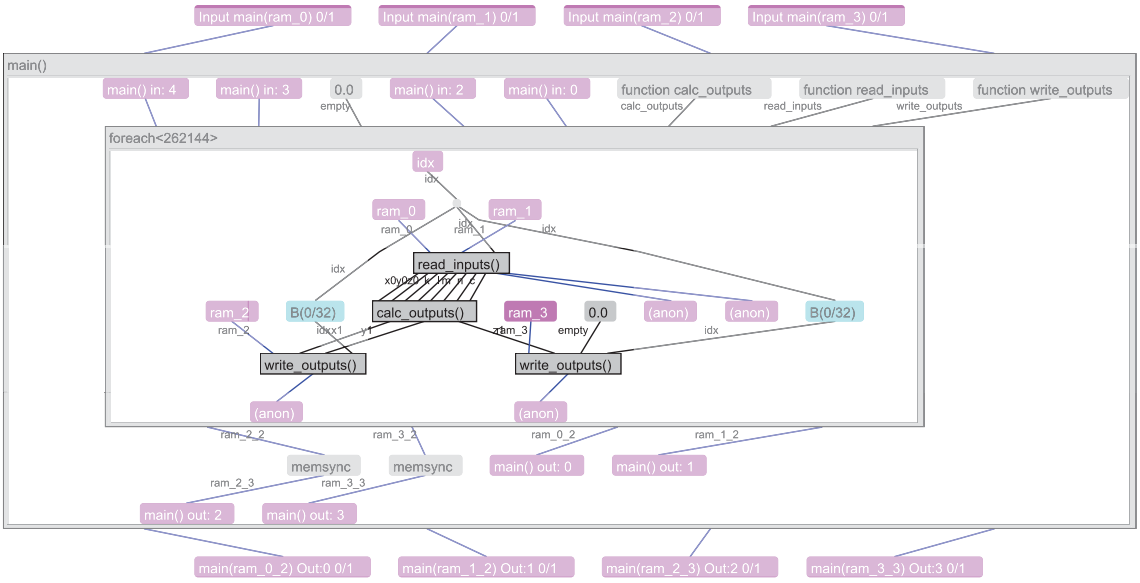

Figure 4.9: Mitrion-C simulation of ray-intersection calculation.

4.4.2 Implementation of the Ray-Intersection Calculation

The primary difference between the ray-intersection calculation and the normal-vector calculation was the fact that the latter calculation requires eight inputs.

The ray-intersection calculation can be simplified into a system of floating point equations, shown below.

$$g = N - c\left(x_0 L + y_0 M + (k+1.0) z_0 N\right) \tag{4.5a}$$

$$h = c\left(x_0^2 + y_0^2 + (k+1.0) z_0^2\right) - 2z_0 \tag{4.5b}$$

$$f = c(1 + kN^2) \tag{4.5c}$$

$$u = \frac{h}{g + \sqrt{g^2 - fh}} \tag{4.5d}$$

$$x_1 = uL + x_0 \tag{4.5e}$$

$$y_1 = uM + y_0 \tag{4.5f}$$

$$z_1 = uN + z_0 \tag{4.5g}$$

This problem therefore requires 11 floating-point additions, 3 subtractions, 19 multiplications, 1 division, and 1 square root. The addition of constants was implemented as a floating-point operation to ensure precision. The numbers of floating-point units implemented reported here reflect the output of the simulator packaged with Mitrion-C.

The FPGA program is listed in appendix A on page 50. Since the FPGA's QDR SRAMs can only hold 4 MB of data each, the number of samples that could be loaded into two memories was half the number that could be stored in the normal-vector calculation (see equation 4.4a). Since

the ray-intersection calculation requires eight inputs, consisting of $8 \times 32 = 256$ bits, it would be impossible to read all eight inputs from just two 64-bit memories (a total of 128 bits) in one computational cycle. There were two options to address this issue: either use four memories for input, or continue to use two SRAMs, but allow more computational time to read the inputs.

Mitrion-C has built into it functions that ensure that a read or write is not attempted of a memory until that memory reports that it is in the ready state. Therefore, it should have been possible to read eight inputs from four memories and simply write the three outputs to QDR SRAMs 2 and 3, overwriting the original inputs. This approach may have worked, but continuing to use only two memories presented a chance to analyze the effects of large numbers of inputs on FPGA processing. There are undoubtedly many applications that would have required more inputs than even four memories could have provided in a single computational cycle. Using two computational cycles would provide valuable insight on the effects on throughput in such a case. Therefore, the eight inputs were stored in QDR SRAMs 0 and 1 in a staggered fashion and the Mitrion-C function *read_inputs()* required two computational cycles to execute. The effects on throughput are discussed in section 5.2 on page 38.

The Mitrion simulation of the ray-intersection calculation is presented in figure 4.9 on the preceding page. The notable difference between this figure and figure 4.8 on page 31 is the fact that the *read_inputs()* function of the ray-intersection calculation is connected to *calc_outputs()* with eight data buses rather than four, which signifies that twice the number of inputs are passed from one function to the next. The fact that the ray-intersection calculation is encapsulated in a single `foreach`loop rather than one loop for each function has no functional significance. The organization of the ray-intersection calculation's host program did not differ significantly from the normal-vector calculation's host program. The host program code is listed in appendix C on page 60

4.5 The Sequential Program

This project set out to compare throughput and power consumption between FPGAs and traditional processors. ANSI-C running on the Opteron 275s was used as the traditional implementation in this project. The GNU-C compiler was used to compile the code and the flag *-O* was used because it instructs the compiler to turn on forms of optimization that do not require any trade-off between speed and space.

The code of the sequential implementations of the ray-intersection calculation and the normal-vector calculation are listed in appendix E on page 74 and F on page 79, respectively. The sequential implementations were designed to mirror the host program implementations as much as possible. Data generation was identical. Time measurements using the *clock()* function only measured the time when actual arithmetic calculations were made. For the sequential programs as well as the host programs the number of samples calculated and the number of times the calculation portion of the program was looped varied in order to ensure that in each case, the time required to process 2^{30} samples was measured.

Figure 4.10 on the next page contrasts the dataflow of the sequential implementation to that of

the FPGA implementation, which uses both an FPGA design and an ANSI-C host program (see figure 4.5 on page 27). The dataflow of the sequential program is clearly much simpler, since the sequential program only interacts with the host memory, while the host program of the FPGA implementation must move data between both the host memory and the FPGA memory. However, the delays incurred by data transfers in the FPGA implementation are negligible compared to overall throughput.

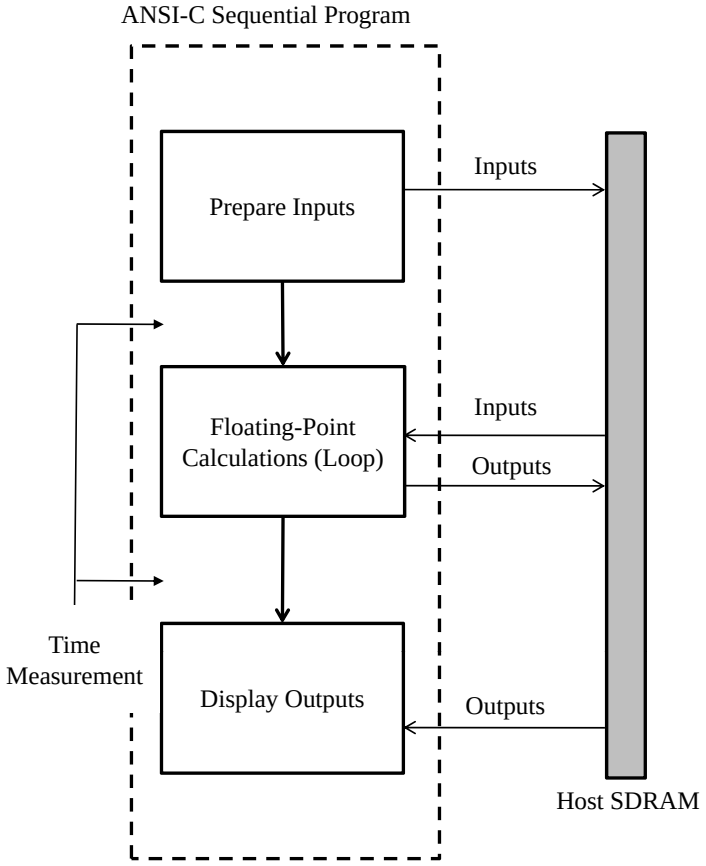

Figure 4.10: Data flow in sequential program.

Chapter 5

Results

This project sought to measure the cost versus benefit of using reconfigurable over conventional processors for HPC. The primary concern for most users of HPC is throughput. Most research comparing FPGAs to CPUs focus on this aspect of performance. However, modern HPC systems generate significant amounts of power and heat. Overheating can even force a system to shut off or reset, which may cause researchers to lose data or have to reprogram experiments. Power and heat are also important to field applications of FPGA processing, where available power cooling abilities may be limited.

Throughput was measured using the *clock()* function of ANSI-C. The measurements were straightforward and are further discussed in section 5.2 on page 38. Power and heat measurements were taken using the Cray XD1's built-in Active Management System (see 4.3 on page 25), a monitoring and control tool for system administrators and end users [33]. This tool was able to isolate and accurately measure power delivered to both the Opteron and FPGA, but the heat resulting could not be isolated with the instrumentation provided with the system. As a result the costs versus benefit analysis achieved in this project compared power to throughput. Resource consumption by the FPGA implementations was also measured, but was not part of the cost-benefit analysis, as explained below, although it would be a suitable focus of further research. In addition, no attempt was made to measure the time taken to implement the calculations, because any such measurement would have required a skilled user of Mitrion-C.

5.1 Resource Consumption

One cost commonly associated with the use of FPGAs is resource consumption. Since FPGAs are reconfigurable, each design that is implemented on one uses different onboard components to implement logic. One part of hardware design, discussed in section 2.2 on page 10, is the loop back to code design after synthesis. VHDL programmers want to maximize the resource consumption of their designs in order to achieve maximum throughput given a set of hardware constraints. The synthesis step of hardware design produces a detailed analysis of expected resource allocation, which the hardware designer uses to make changes to a design.

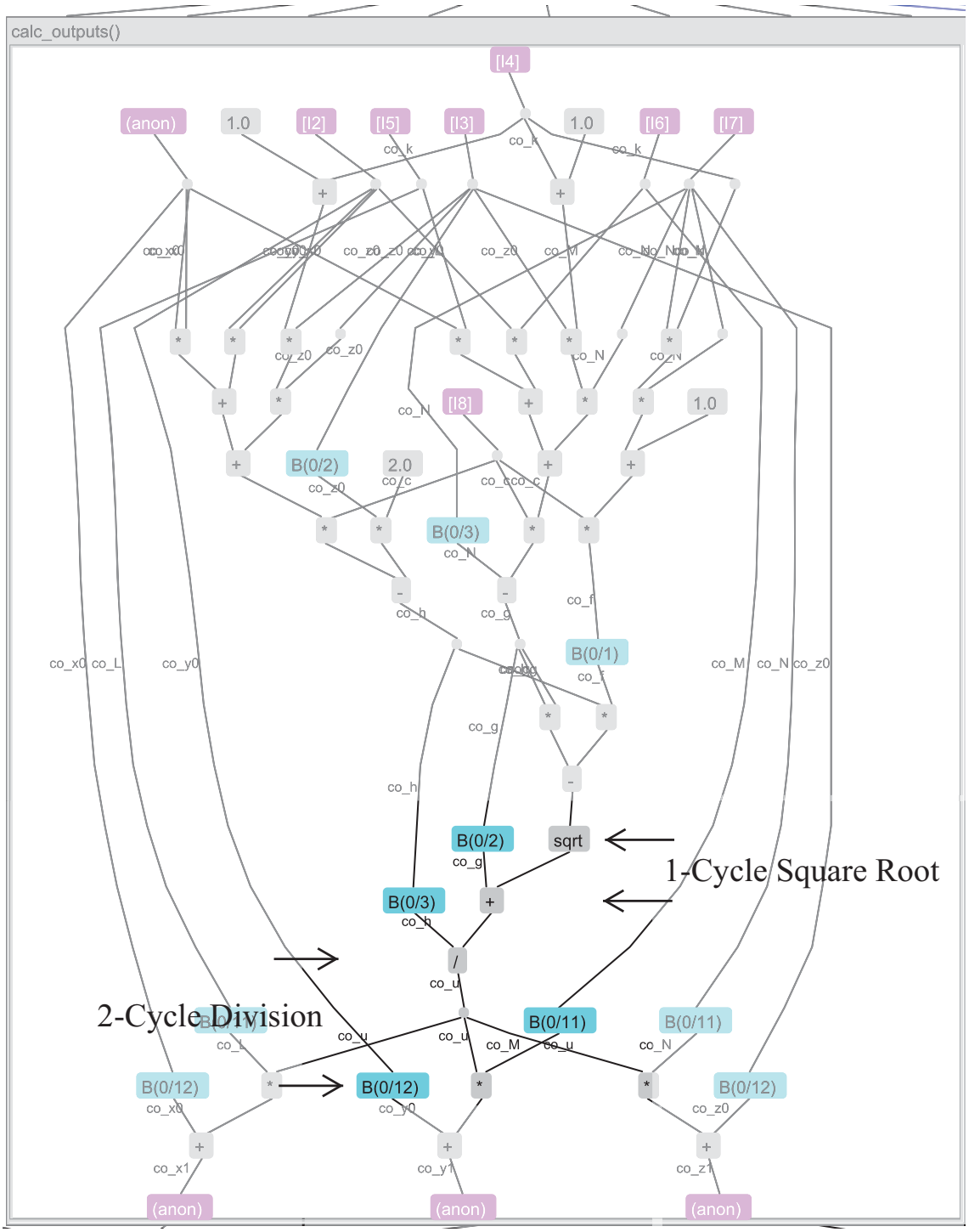

Figure 5.1: Mitrion-C simulation of *calc_ouputs()* function of ray-intersection simulation.

Table 5.1: Ray-intersection resource consumption.

Resource (total)	Implemented (Percent)
Slices (23 616)	19 044 (81 %)
Flip Flops (47 232)	26 508 (56 %)
4-input LUTs (47 232)	26 250 (56 %)
Block RAMs (232)	25 (11 %)
Multipliers (232 18x18)	72 (31 %)

Table 5.2: Normal-vector resource consumption.

Resource (total)	Implemented (Percent)
Slices (23 616)	16 571 (70 %)
Flip Flops (47 232)	21 670 (46 %)
4-input LUTs (47 232)	20 466 (43 %)
Block RAMs (232)	23 (10 %)
Multipliers (232 18x18)	48 (21 %)

Although Mitrion-C allows explicit definition of parallelism, it does not permit the same fine-grained control over a design's resource use as a traditional HDL does. section 2.5 on page 15 describes the many considerations that must be taken into account when a hardware designer creates a design. In Mitrion-C, scheduling is automated and opaque to the user. Mitrion's simulator shows when arithmetic operations are begun, giving an indication of how scheduling is done. In figure 5.1 on the previous page, each interval of space in the vertical direction corresponds with a unit of time, that is, one computational cycle. Analysis of this flow shows that the Mitrion compiler uses As-Late-As-Possible scheduling, described in section 2.5 on page 15 and illustrated in figure 2.5 on page 15. However, the simulation output also seems to indicate that with the exception of division, every arithmetic operator can be completed in just one computational cycle. Experience with other floating-point operator implementations makes it seem unlikely this is actually the case.

The simulator output also shows no use of modulo scheduling, described in section 2.5 on page 15. Implementation of modulo scheduling would lead to multiple numbers of floating-point units being implemented based on need. However, the simulation output indicates that one floating-point operator was implemented for each operation completed. The ability of the Mitrion Virtual Processor to generate one output per computational cycle to a complex problem such as the ray-intersection calculation using the schedule that is illustrated in figure 5.1 on the previous page seems highly unlikely. It seems more probable that the simulator merely produces a simple ALAP-scheduled representation of the logic for debugging purposes, and that more complex scheduling and optimization is not visible to the user. The VHDL code the Mitrion compiler generates is complex and not designed to be analyzed by the end user. Attempts to do so were unsuccessful.

Table 5.3: Ray-intersection throughput measurements.

	Opteron 275	Virtex-II Pro
Rays Traced	1 073 741 824	
Time (s)	219.54	21.49
Throughput (rays/s)	4.891×10^6	4.996×10^7
Speedup	—	10.21×
	—	921%

Tables 5.1 on the preceding page and 5.2 on the previous page report the resources consumed by the ray-intersection and normal-vector calculations, respectively. Xilinx defines slices as the basic configurable logic unit within an FPGA. Each one contains two 4-input lookup tables (LUTs) and two flip-flops. The lookup tables are used to implement simple logic equations and the flip-flops are used to hold the outputs of the lookup tables when that is required. In both designs, well over 50% of available slices were used by the designs. It was therefore not possible using Mitrion-C to implement multiple instances of the designs in order to obtain greater throughput because doing so would have required twice as many slices.

5.2 Throughput Measurement

As discussed in section 4.4 on page 29, the FPGA implementation of the ray-intersection calculation could process 262 144 samples before needing new inputs while the normal-vector calculation could process 524 288 samples. The ray-intersection calculation was iterated by the host program 8192 times and the normal-vector calculation 4096 times. Each time measurement therefore measured the time needed to process 1 073 741 824=2^{30} rays.

The time functions built into ANSI-C were used for time measurement. The `clock()` function returns the system time given in *clock ticks* relative to an arbitrary reference time. The time was measured before starting the FPGA program and again once all iterations were complete. By subtracting the start value from the end value and dividing by the macro `CLOCKS_PER_SEC`, which stores the number of clock ticks per second measured by `clock()`, time elapsed in seconds could be calculated. For the sequential implementations of the two calculations, the time was measured before the loop that encapsulated the arithmetic portion of the program began and again after it completed. The results of the measurements are presented in tables 5.3 and 5.4 on the following page.

The maximum amount of speedup achieved—10.67×— seemed reasonable compared to past research. The results also seemed to show that the throughput indicated during simulation—that is, one full calculation delivered per cycle in the case of the normal-vector calculation—was correctly implemented. The Mitrion Virtual Processor has been shown to run on the Cray XD1 at 100 MHz [26]. Therefore, the minimum time t required to complete 1 073 741 824 calculations can be found using the equation:

Table 5.4: Normal-vector throughput measurements.

	Opteron 275	Virtex-II Pro
Rays Traced	1 073 741 824	
Time (s)	114.79	10.75
Throughput (rays/s)	9.354×10^6	9.988×10^7
Speedup	—	10.67×
	—	967%

$$t = \frac{1\text{s}}{100 \times 10^6 \,\text{clock cycles}} \times \frac{1\,\text{clock cycle}}{\text{calculation}} \times 1\,073\,741\,824\,\text{calculations} = 10.74\text{s} \tag{5.1a}$$

As discussed in section 4.4.2 on page 32, the FPGA implementation of the ray-intersection calculation required two clock cycles for each calculation. If equation 5.1a were applied to the ray-intersection calculation, the minimum time t would equal 21.47 seconds. The expected value of elapsed time for both the normal-vector and ray-intersection calculations were essentially equal to the measured values listed in tables 5.4 and 5.3 on the previous page. This result supports the likelihood that modulo scheduling is automatically implemented by the Mitrion compiler because only a schedule with a modulus of one could have achieved maximum theoretical throughput.

The fact that the ray-intersection calculation achieved the same speedup as the normal-vector calculation was surprising. As described in section 5.3 on the preceding page, the implementation of the ray-intersection calculation only used two QDR SRAMs, though theoretically it could have used four. The fact that the throughput of the sequential program decreased about the same amount as the FPGA implementation seems to indicate that the sequential program was also limited by memory bandwidth. However, the results suggest that the FPGA implementation should have been able to double its throughput had four memories been used instead of two. If such a implementation could be developed, a greater than 20× speedup seems possible.

5.3 Power Measurement

Power consumption was measured using Cray's Hardware Supervisory Subsystem (HSS) software, which runs on the management processor of each Cray XD1 chassis and is designed to monitor the health of the system [33]. The HSS reports the voltage supplied to the regulators of each node and the current supplied by the regulators to individual components of the node, including the Opterons and FPGAs. Because both the Opterons and FPGAs draw power even when idle, monitoring total power (in watts) was of greatest interest. For this reason power measurements were taken both on nodes with an FPGA present and on nodes without. In all, five power levels for each of the two calculations implemented were measured:

1. a node without an FPGA while idle,

2. a node without an FPGA while running the sequential implementation,
3. a node with an FPGA while idle,
4. a node with an FPGA while running the sequential implementation, and
5. a node with an FPGA while running the FPGA implementation.

Table 5.5: Ray-intersection power measurements.

Node Type	Implementation	Total Power (watts)
No FPGA	Idle	102.65
FPGA	Idle	130.94
No FPGA	Sequential Only	110.87
FPGA	Sequential Only	139.57
FPGA	FPGA	142.87

Table 5.6: Normal-vector power measurements.

Node Type	Implementation	Total Power (watts)
No FPGA	Sequential Only	111.84
FPGA	Sequential Only	139.84
FPGA	FPGA	143.66

Three sets of 100 samples each were measured at 2 s intervals. The full datasets are displayed in figures 5.2 on the next page through 5.9 on page 45. By comparing the mean values and standard deviations between sets, it was found that power was independent of time, that is, stationary at the scale of the measurements. Tables 5.5 and 5.6 present the mean values of the findings across all 300 samples in each case. Idle power measurements are omitted from Table 5.6 because they were equal to the values presented in Table 5.5.

In the case of the ray-intersection calculation, the Virtex-II Pro implementation required $1.285\times$ the power of the sequential program running on a node with no FPGA (a 28.5% increase) and $1.027\times$ the power of the sequential program running on a node with an FPGA (a 2.7% increase). For the normal-vector calculation, the FPGA implementation required $1.259\times$ and $1.011\times$ the power of the sequential program (increases of 25.9% and 1.1%), respectively.

The background power required for any system will vary based on the particular operating system or other processes that are running in addition to the calculation of interest. Processing unit power can be isolated from background power by calculating the ratio

$$\frac{P_{\text{FPGA}} - P_{\text{FPGA,IDLE}}}{P_{\text{SEQ}} - P_{\text{SEQ,IDLE}}} \tag{5.2}$$

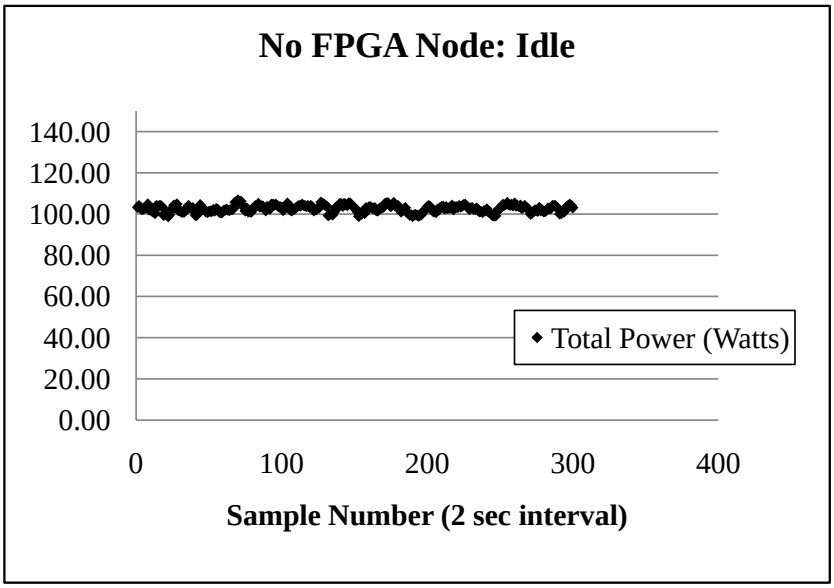

Figure 5.2: Background power measurements of a node without an FPGA.

where P_{SEQ} is the power consumed by the sequential program in a node with no FPGA attached, $P_{\text{SEQ,IDLE}}$ is the power consumed in the same node when the sequential program is not executing (although the operating system's instructions will still be executing in that node), P_{FPGA} is the power consumed by the parallel hardware design in an FPGA in a node with the FPGA attached, and $P_{\text{FPGA,IDLE}}$ is the power consumed by that same node when the FPGA does not contain the parallel design, so is idling. The result was 1.240× power consumption (24.0% increase) for the ray-intersection calculation and 1.451× (45.1% increase) for the normal-vector calculation.

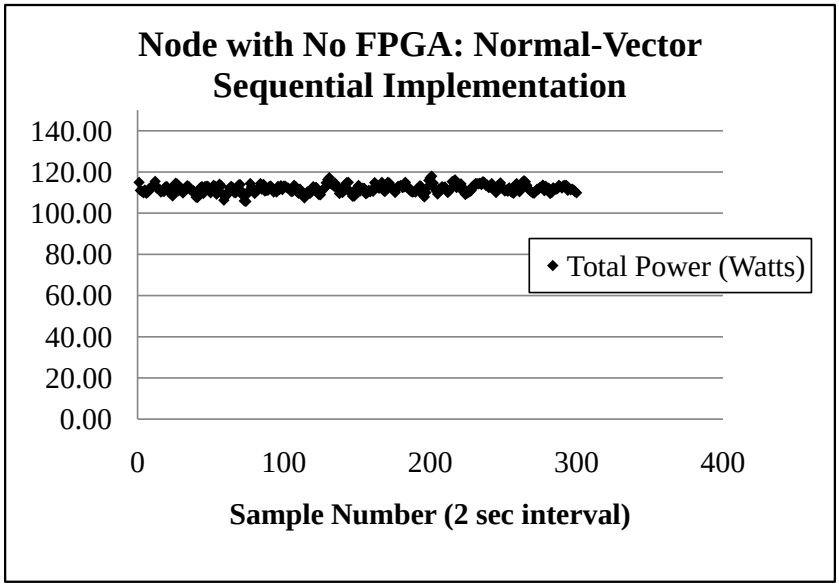

Figure 5.3: Normal-vector calculation implemented with only an Opteron 275 on a node without an FPGA.

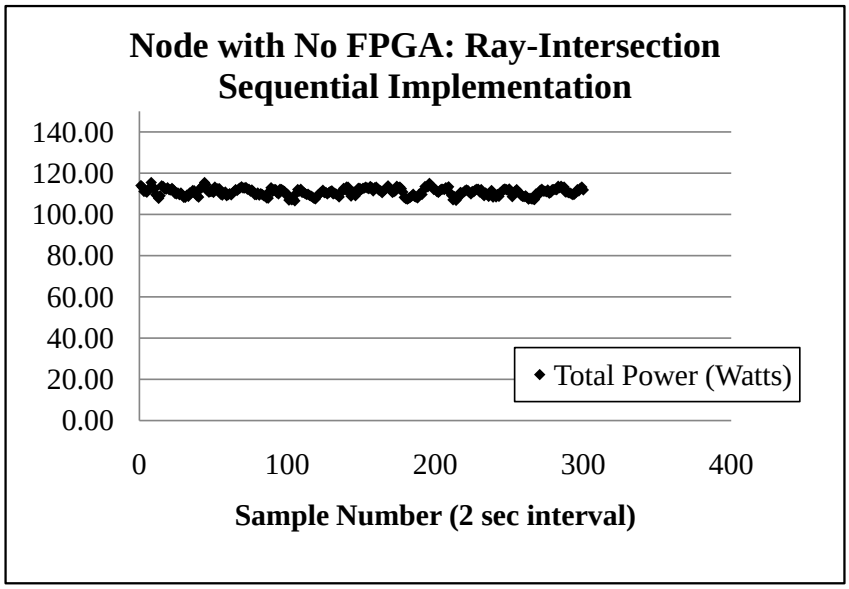

Figure 5.4: Ray-intersection calculation implemented with only an Opteron 275 on a node without an FPGA.

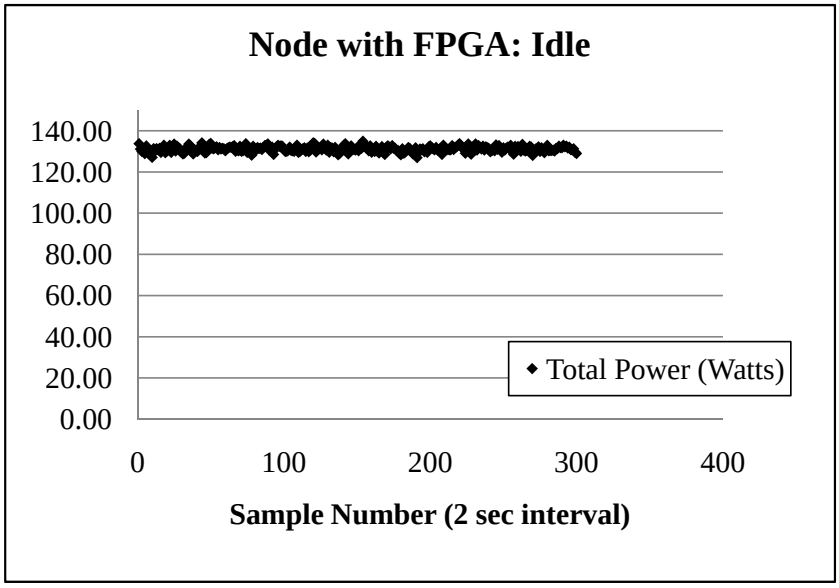

Figure 5.5: Background power measurements of a node with an FPGA.

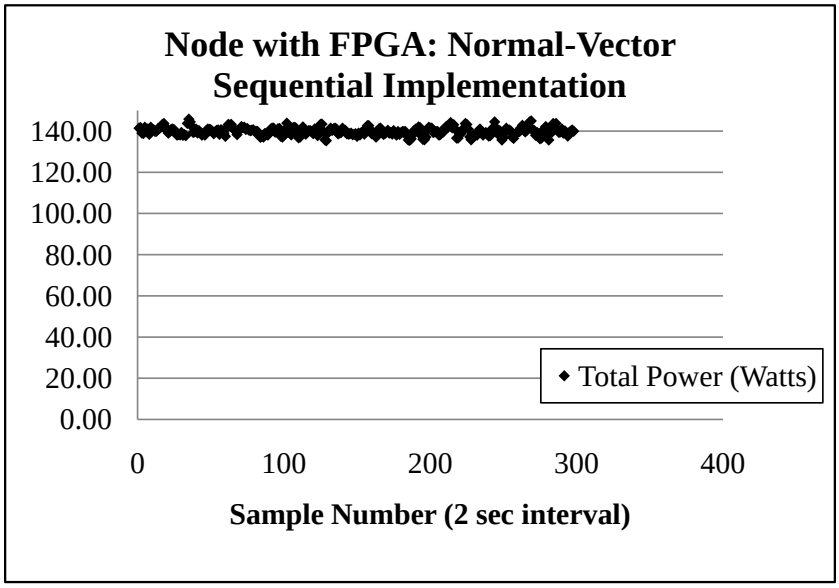

Figure 5.6: Normal-vector calculation implemented with only an Opteron 275 on a node with an FPGA.

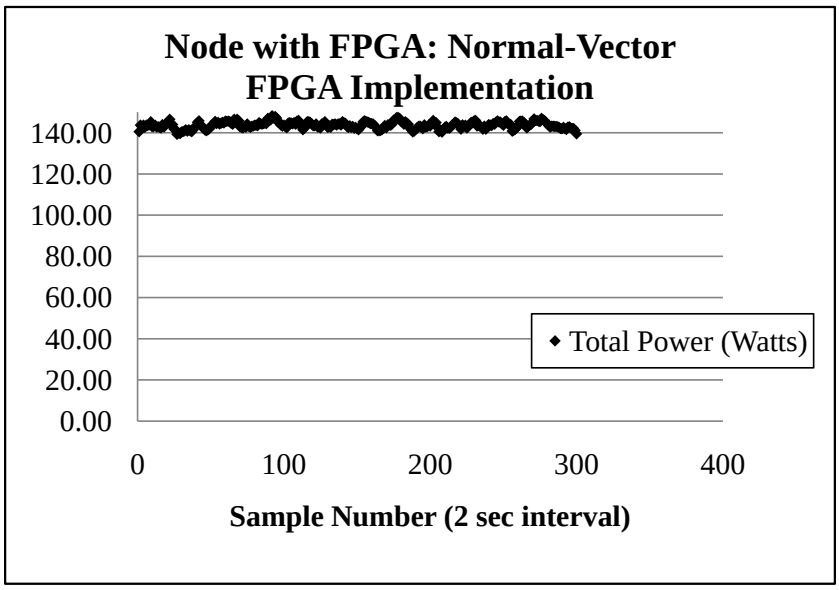

Figure 5.7: Normal-vector calculation implemented with a Virtex-II Pro and an Opteron 275.

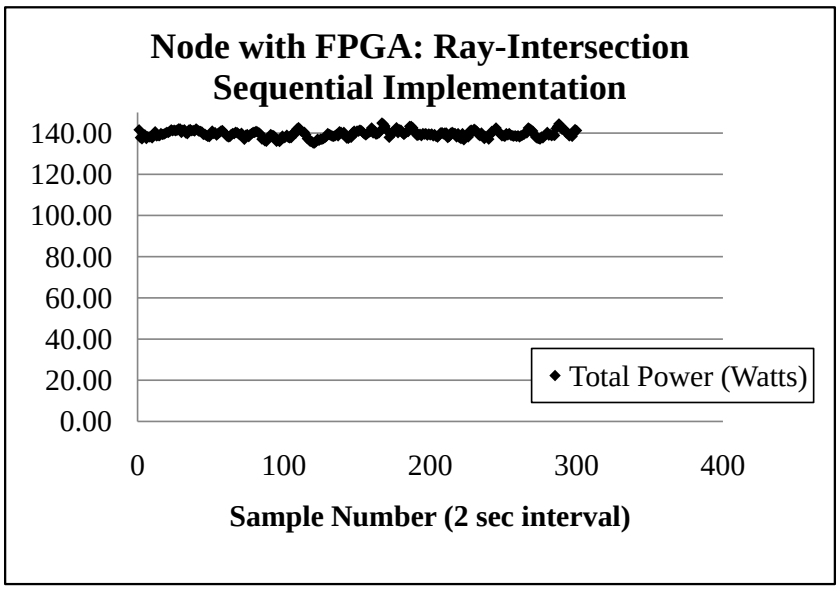

Figure 5.8: Ray-intersection calculation implemented with only an Opteron 275 on a node with an FPGA.

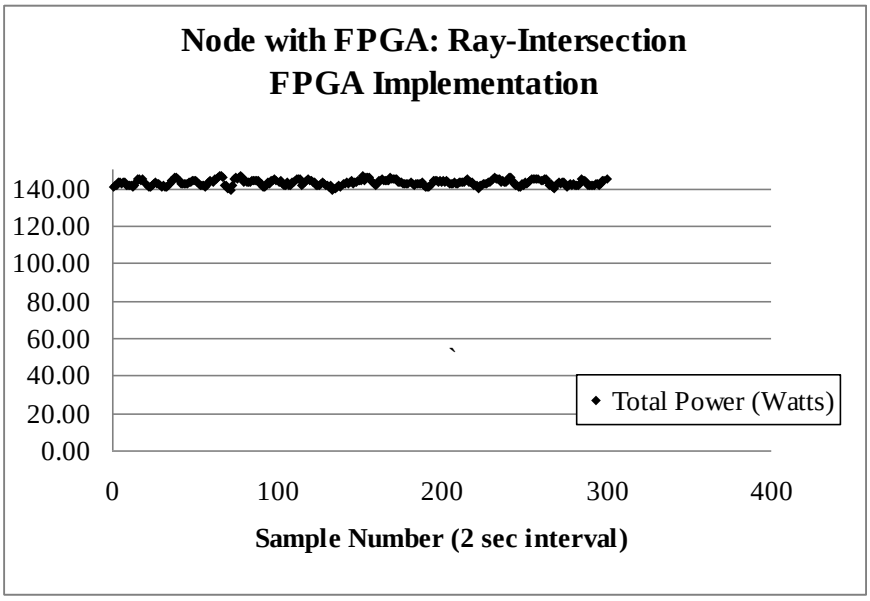

Figure 5.9: Ray-intersection calculation implemented with a Virtex-II Pro and an Opteron 275.

Chapter 6

Conclusion

In this paper, the acceleration of two portions of the optical simulation of NASA's Moderate Resolution Imaging Spectroradiometer was presented. Mitrion-C HLL was used to implement hardware designs on Virtex-II Pro FPGAs. A functionally equivalent program was written using ANSI-C and implemented on an Advanced Micro Devices Opteron 275 processor.

Throughput and power of all implementations were measured on the Cray-XD1 supercomputer. Recent marketing literature from Mitrionics AB—the developer of Mitrion-C—has claimed the ability of FPGAs to process at speeds of up to 100 times faster than sequential processors and to use only 2% as much power when operating at the same speed as sequential processors [30]. The maximum speedup measured in this project was 10.67×, or a 967% increase. This speedup was measured using only two of the FPGA's four memories for input. It is predicted that the measured speedup would have been doubled had all four memories been used. However, the feasibility of such an implementation from a resource and power consumption standpoint are unknown.

The maximum power increase required to run an FPGA was measured to be 45.1% when power consumed by the processing unit was isolated. However, many researchers may be more interested in total power consumption because of overall heat and cost limits. Taking background power into account, the maximum power increase required to run an FPGA was measured to be 28.5%. Throughput and power are presented separately as benefit and cost because different applications may weight different factors more heavily, and so no one direct comparison would be comprehensive.

The results showed that floating-point operations using FPGAs offer significant speedup over sequential processor implementations without excessive additional power consumption. It was also shown that high-level languages such as Mitrion-C can reduce development times and the need for extensive experience with hardware design and still achieve efficient FPGA use.

The greatest disadvantage observed in this research to using FPGAs for high-performance computing was the need for a sequential host program to feed new data to the FPGA. Even when the sequential program was not contributing directly to the calculations, it continued to consume a significant amount of power. There are two ways to avoid this problem: (1) eliminate the sequential processor and find another way to feed data to the FPGA or (2) use the sequential processor both to feed data to the FPGA and to perform some of the calculations in parallel with the FPGA.

Bibliography

[1] Cray, Inc., "Cray XD1 datasheet," Cray Inc., Tech. Rep., June 2005. [Online]. Available: http://www.cray.com/downloads/Cray_XD1_Datasheet.pdf

[2] Naval Research Laboratory, "Department of Defense (DOD) high performance computing-modernization plan," Dec. 2006. [Online]. Available: http://www.cmf.nrl.navy.mil/CCS/hpc-nrl.html

[3] R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin, and C. Kitchen, "An overview of FPGAs and FPGA programming; initial experiences at Daresbury," *Computational Science and Engineering Department*, Nov. 2006.

[4] IEEE, "IEEE standard for binary floating-point arithmetic, ANSI-IEEE std 754-1985," IEEE Standards Board, Tech. Rep., 1985.

[5] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pp. 155–162, 19-21 Apr 1995.

[6] P. Belanović and M. Leeser, "A library of parameterized floating point modules and their use," in *12th International Conference on Field Programmable Logic and Application, FPL 2002*, Montpellier, France, September 2002, pp. 657–666. [Online]. Available: citeseer.ist.psu.edu/belanovic02library.html

[7] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier, "A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs," in *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2002, pp. 50–55.

[8] C. B. Cameron, "Parallel ray tracing using the message passing interface (MPI)," *IEEE Trans. Instrum. Meas.*, vol. 57, no. 2, pp. 228–234, Feb. 2008.

[9] S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A tutorial," 1996. [Online]. Available: citeseer.ist.psu.edu/brown96architecture.html

[10] M. Barr, "How programmable logic works." Oct. 2007. [Online]. Available: http://www.netrino.com/Articles/ProgrammableLogic/index.php

[11] A. Dellson, G. Sandberg, and S. Möhl, "Turning FPGAs into supercomputers." *Cray User Group*, 2006.

[12] Y.-C. Hsu and Y.-L. Jeang, "Pipeline scheduling techniques in high-level synthesis," *ASIC Conference and Exhibit, 1993. Proceedings., Sixth Annual IEEE International*, pp. 396–403, 27 Sep-1 Oct 1993.

[13] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *SIGMICRO Newsl.*, vol. 12, no. 4, pp. 183–198, 1981.

[14] B. Fagin and C. Renard, "Field-programmable gate arrays and floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 3, pp. 365–367, Sept. 1994.

[15] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs," in *FPGAs for Custom Computing Machines*, Apr. 1996, pp. 107–116.

[16] A. A. Gaffar, W. Luk, P. Y. Cheung, N. Shirazi, and J. Hwang, "Automating customization of floating-point designs," in *International Conference on Field-Programmable Logic and Applications*, Aug. 2002.

[17] A. A. Gaffar, O. Mencer, W. Luk, P. Cheung, and N. Shirazi, "Floating point bitwidth analysis via automatic differentiation," in *International Conference on Field-Programmable Technology*, 2002.

[18] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," *ACM/SIGDA Twelfth ACM International Symposium on Field-Programmable Gate Arrays (FPGA 2004)*, 2004.

[19] G. Genest, R. Chamberlain, and R. Bruce, "Programming an FPGA-based super computer using a C-to-VHDL compiler: DIME-C," in *Second NASA/ESA Conference on Adaptive Hardware and Systems*, Aug. 2007, pp. 280–286.

[20] G. H. Spencer and M. V. R. K. Murtry, "General ray-tracing procedure," *J. Opt Soc. Ameri.*, vol. 52, no. 6, pp. 652–678, June 1962.

[21] C. Cameron, R. Rodriguez, N. Padgett, E. Waluschka, and S. Kizhner, "Optical ray tracing using parallel processors," *IEEE Trans. Instrum. Meas.*, vol. 54, no. 1, pp. 87–97, Feb. 2005.

[22] C. Cameron, R. Rodriguez, N. Padgett, E. Waluschka, S. Kizhner, G. Colon, and C. Weeks, "Fast optical ray tracing using parallel DSPs," *IEEE Trans. Instrum. Meas.*, vol. 55, no. 3, pp. 801–808, June 2006.

[23] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conf. Proc.*, vol. 30, 1967, pp. 483–485.

[24] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From high-level language to hardware circuitry," *Field Programmable Logic and Applications, 2006. FPL 2006. International Conference on*, vol. 40, no. 3, pp. 28–37, 2007.

[25] J. Koo, A. Evans, and W. Gross, "Accelerating a medical 3D brain MRI analysis algorithm using a high-performance reconfigurable computer," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 11–16, 27-29 Aug. 2007.

[26] J. J. Koo, D. Fernandez, A. Haddad, and W. J. Gross, "Evaluation of a high-level-language methodology for high-performance reconfigurable computers," *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pp. 30–35, 9-11 July 2007.

[27] S. D. Landy and A. S. Szalay, "Bias and variance of angular correlation functions," *Astrophys. J.*, vol. 412, pp. 64–71, July 1993. [Online]. Available: http://dx.doi.org/10.1086%2F172900

[28] V. V. Kindratenko, R. J. Brunner, and A. D. Myers, "Mitrion-C Application Development on SGI Altix 350/RC100," *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pp. 239–250, 23-25 April 2007.

[29] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. Newby, "Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study," *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, pp. 99–106, 28-26 Feb. 2007.

[30] Mitrionics AB, "Accelerate your applications—unleash the massive performance of FPGAs." [Online]. Available: http://www.mitrion.com/press/Mitrion_product_brief.pdf

[31] S. Mohl, "The Mitrion-C programming language," *Mitrionics Inc.*, 2006. [Online]. Available: http://www.mitrionics.com/

[32] M. J. Kidger, *Fundamental Optical Design*. Fundamental Optical Design by Michael J. Kidger Bellingham, WA: SPIE-The International Society for Optical Engineering, 2002, 2002.

[33] Cray Inc., "Cray XD1 glossary," *Cray Inc.*, 2005.

Appendix A

Mitrion-C Code of Ray-Intersection Calculation

```
Mitrion-C 1.3;

// Options: -cpp

// Decimal value 4*16^4 = 262144
#define NUM_SAMPLES 0x40000

// Memories must be defined with 64-bit width and 20-bit address space.
#define ExtRAM mem bits:64 [0x100000]

// Define "Float" as a 32-bit single-precision floating-point number.
#define Float float:24.8


// This function reads the eight inputs from the QDR SRAMs 0 and 1.
(Float, Float, Float, Float, Float, Float, Float, Float, ExtRAM, ExtRAM)
   read_inputs(ri_ram_0, ri_ram_1, ri_idx)
{
```

```
    ri_offset = 2*ri_idx;

    bits:64 ri_b64_0;
    bits:64 ri_b64_1;
    bits:64 ri_b64_2;
    bits:64 ri_b64_3;

    // By using the correctly memory references, Mitrion can ensure
    // that a QDR SRAM is ready before data is read.
    (ri_b64_0, ri_ram_0_1) = _memread(ri_ram_0, ri_offset);
    (ri_b64_1, ri_ram_1_1) = _memread(ri_ram_1, ri_offset);
    (ri_b64_2, ri_ram_0_2) = _memread(ri_ram_0_1, ri_offset+1);
    (ri_b64_3, ri_ram_1_2) = _memread(ri_ram_1_1, ri_offset+1);

    // Combine 4 64-bit words into a single 256-bit word.
    bits:256 ri_b256 = [ri_b64_0, ri_b64_1, ri_b64_2, ri_b64_3];

    // Convert the 256-bit string into a list of 4 32-bit words.
    Float [8] b_to_f = ri_b256;

    // Convert the 32-bit words into floating-point representation.
    Float ri_a = b_to_f[0];
    Float ri_b = b_to_f[1];
    Float ri_c = b_to_f[2];
    Float ri_d = b_to_f[3];

    Float ri_e = b_to_f[4];
    Float ri_f = b_to_f[5];
    Float ri_g = b_to_f[6];
    Float ri_h = b_to_f[7];

    // Statements to make sure all memory reads are complete.
```

```
        ri_ram_0_3 = _wait(ri_ram_0_2);
        ri_ram_1_3 = _wait(ri_ram_1_2);

} (ri_a, ri_b, ri_c, ri_d, ri_e, ri_f, ri_g, ri_h, ri_ram_0_3, ri_ram_1_3);

// This function implements the series of arithmetic
// operations associated with the ray-intersection calculation.
(Float, Float, Float) calc_outputs(co_x0, co_y0, co_z0, co_k, co_L, co_M, co_N,
   co_c)
{
        co_g = co_N-co_c*(co_x0*co_L+co_y0*co_M+(co_k+1.0)*co_z0*co_N);
        co_h = co_c*(co_x0*co_x0+co_y0*co_y0+(co_k+1.0)*co_z0*co_z0)-2.0*co_z0;
        co_f = co_c*(1.0+co_k*co_N*co_N);
        co_u = (co_h)/(co_g+_sqrt(co_g*co_g-co_f*co_h));
        co_x1 = co_u*co_L+co_x0;
        co_y1 = co_u*co_M+co_y0;
        co_z1 = co_u*co_N+co_z0;

// The three outputs are the coordinates of the point of intersection.
} (co_x1, co_y1, co_z1);


// This function writes two outputs to a QDR SRAM.
(ExtRAM) write_outputs(wo_ram_0, wo_out_0, wo_out_1, wo_offset)
{
        // Combine two 32-bit floating-point values into a list.
        Float[2] f_to_b = [wo_out_0, wo_out_1];

        // Convert the list into a 64-bit word.
        bits:64 wo_b64 = f_to_b;

        // Write the 64-bit word to the target QDR SRAM.
```

```
81          wo_ram_0_1 = _memwrite(wo_ram_0, wo_offset, wo_b64);

83          // Waits until write is complete. This command ensures
            // the memory is ready before another write is attempted
85          wo_ram_0_2 = _wait(wo_ram_0_1);
} (wo_ram_0_2);
87
// This is the main program. For the Cray XD1, the main program
89 // must take four 64-bit external memories as input. It also must
   // return four external memories as output. The memories are
91 // passed as references.
(ExtRAM, ExtRAM, ExtRAM, ExtRAM)
93 main(ExtRAM ram_0, ExtRAM ram_1, ExtRAM ram_2, ExtRAM ram_3)
{
95          // Used to fill the 32 unused bits in ram 3.
            Float empty = 0.0;
97
            Float<NUM_SAMPLES> final_x1;
99          Float<NUM_SAMPLES> final_y1;
            Float<NUM_SAMPLES> final_z1;
101
            // This loop executes each of the functions within in parallel.
103         // It iterates across each sample, for a total of 262144 times.
            // The Mitrion compiler resolves data dependencies within the
105         // loop, automatically allocating resources for maximum
            // throughput.
107         (ram_2_2, ram_3_2, ram_0_2, ram_1_2, final_x1, final_y1, final_z1) =
               foreach(idx in <0.. NUM_SAMPLES-1>)
            {
109                 (x0, y0, z0, k, l, m, n, c, ram_0_1, ram_1_1) = read_inputs(ram_0,
                        ram_1, idx);
                    (x1, y1, z1) = calc_outputs(x0, y0, z0, k, l, m, n, c);
```

```
111                 ram_2_1 = write_outputs(ram_2, x1, y1, idx);
                    ram_3_1 = write_outputs(ram_3, z1, empty, idx);
113             } (ram_2_1, ram_3_1, ram_0_1, ram_1_1, x1, y1, z1);

115             // Passes the last memory reference for these rams so the
                // final reference is passed out of the main program.
117             ram_2_3 = _wait(ram_2_2);
                ram_3_3 = _wait(ram_3_2);
119
    } (ram_0_2, ram_1_2, ram_2_3, ram_3_3);
```

Appendix B

Mitrion-C Code of Normal-Vector Calculation

.

```
Mitrion-C 1.3;

// Options: -cpp

// = 8*16^4 = 524288
#define NUM_SAMPLES 0x80000

// Memories must be defined with 64-bit width and 20-bit address space.
#define ExtRAM mem bits:64 [0x100000]

// Define "Float" as a 32-bit single-precision floating-point number.
#define Float float:24.8


// This function reads the four inputs from the QDR SRAMs 0 and 1.
(Float, Float, Float, Float) read_inputs(ri_ram_0, ri_ram_1, ri_offset)
{
        // Read in a 64-bit word.
```

```
        bits:64 ri_b64_0 = _memread(ri_ram_0, ri_offset);
        bits:64 ri_b64_1 = _memread(ri_ram_1, ri_offset);

        // Combine the two 64-bit words into a 128-bit word.
        bits:128 ri_b128 = [ri_b64_0, ri_b64_1];

        // Convert the 128-bit word into a list of of 4 32-bit words.
        Float [4] b_to_f = ri_b128;

        // Convert the 32-bit words into floating-point representation.
        Float ri_x = b_to_f[0];
        Float ri_y = b_to_f[1];
        Float ri_u = b_to_f[2];
        Float ri_c = b_to_f[3];

} (ri_x, ri_y, ri_u, ri_c);


// This function implements the series of arithmetic
// operations associated with the normal-vector calculation.
(Float, Float, Float) calc_outputs(co_x, co_y, co_u, co_c)
{
        co_v = co_u * (co_x * co_x + co_y * co_y);
        co_a = _sqrt(1 - co_v);
        co_p = 1.0 + co_a;
        co_q = co_a * co_p;
        co_r = co_p * co_q;
        co_s = 2.0 * co_q;
        co_w = co_c/co_r;
        co_b = co_w * (co_s + co_v);
        co_dx = co_b * co_x;
        co_dy = co_b * co_y;
```

```
        co_e = _sqrt(co_dx * co_dx + co_dy * co_dy + 1);
        co_f = 1.0 / co_e;
        co_fdx = co_f * co_dx;
        co_fdy = co_f * co_dy;

// The three outputs are the three components of the normal vector.
} (co_f, co_fdx, co_fdy);


// This function writes two outputs to a QDR SRAM.
(ExtRAM) write_outputs(wo_ram_0, wo_out_0, wo_out_1, wo_offset)
{
        // Combine two 32-bit floating-point values into a list.
        Float[2] f_to_b = [wo_out_0, wo_out_1];

        // Convert the list into a 64-bit word.
        bits:64 wo_b64 = f_to_b;

        // Writes the 64-bit word to the target QDR SRAM.
        wo_ram_0_1 = _memwrite(wo_ram_0, wo_offset, wo_b64);

        // Waits until write is complete. This command ensures
        // the memory is ready before another write is attempted
        wo_ram_0_2 = _wait(wo_ram_0_1);
} (wo_ram_0_2);


// This is the main program. For the Cray XD1, the main program
// must take four 64-bit external memories as input. It also must
// return four external memories as output. The memories are
// passed as references.
(ExtRAM, ExtRAM, ExtRAM, ExtRAM)
```

```
main(ExtRAM ram_0, ExtRAM ram_1, ExtRAM ram_2, ExtRAM ram_3)
{
        // Used to fill the 32 unused bits in ram 3.
        Float empty = 0.0;

        Float<NUM_SAMPLES> final_f;
        Float<NUM_SAMPLES> final_fdx;
        Float<NUM_SAMPLES> final_fdy;

        (final_x, final_y, final_u, final_c) = foreach(idx in <0.. NUM_SAMPLES-1>)
        {
                // Read 4x524288 inputs.
                (x, y, u, c) = read_inputs(ram_0, ram_1, idx);
        } (x, y, u, c);

        (final_f, final_fdx, final_fdy) = foreach(x_0, y_0, u_0, c_0, idx in
           final_x, final_y, final_u, final_c, <0.. NUM_SAMPLES-1>)
        {
                // The inputs are passed to calc_outputs as soon as they
                // are read; there is no need to wait for all 524288 memory
                // accesses to complete. The code is run in parallel.
                (f, fdx, fdy) = calc_outputs(x_0, y_0, u_0, c_0);
        } (f, fdx, fdy);

        (ram_2_2, ram_3_2) = foreach(f_0, fdx_0, fdy_0, idx in final_f, final_fdx,
            final_fdy, <0.. NUM_SAMPLES-1>)
        {
                // Writes f and fdx to ram 2.
                ram_2_1 = write_outputs(ram_2, f_0, fdx_0, idx);

                // Writes fdy and a filler to ram 3. The filler is
                // ignored by the host program.
```

```
                ram_3_1 = write_outputs(ram_3, fdy_0, empty, idx);
        }(ram_2_1, ram_3_1);

        // Passes the last memory reference for these rams so the
        // final reference is passed out of the main program.
        ram_2_3 = _wait(ram_2_2);
        ram_3_3 = _wait(ram_3_2);

} (ram_0, ram_1, ram_2_3, ram_3_3);
```

Appendix C

ANSI-C Host Code of Ray-Intersection Calculation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mithal.h"
#include "mithal_gen.h"
#include "float.h"

#define NUM_SAMPLES 131072

FPGA *f;
Processor *p;
float *ram_a;
float *ram_b;
float *ram_c;
float *ram_d;

union hex_float {
    float f;
```

```
19      long l;
};

int init_fpga()
{
        STATUS s;

        //The following code is provided by Mitrionics for host-FPGA interface.
        // Allocate a FPGA
        f = mitrion_fpga_allocate("");
        if(f == NULL) {
        fprintf(stderr, "Could not allocate a FPGA\n");
        exit(1);
        }

        // Create Mitrion processor
        p = mitrion_processor_create("top.bin.ufp");
        if(f == NULL) {
        fprintf(stderr, "Could not create Mitrion processor\n");
        exit(1);
        }

        // Load the Mitrion processor onto the FPGA
        s = mitrion_fpga_load_processor(f, p);
        if(s != OK) {
        fprintf(stderr, "Could not load the Mitrion processor onto the FPGA\n");
        exit(1);
        }

        // Associate virtual memory spaces with QDR SRAM memory addresses.
        ram_a = (float*)mitrion_processor_reg_buffer(p, "ram_0", NULL, NUM_SAMPLES
            *2*sizeof(float)*8, WRITE_DATA);
```

```
51          ram_b = (float*)mitrion_processor_reg_buffer(p, "ram_1", NULL, NUM_SAMPLES
                *2*sizeof(float)*8, WRITE_DATA);

53          ram_c = (float*)mitrion_processor_reg_buffer(p, "ram_2", NULL, NUM_SAMPLES
                *2*sizeof(float)*8, READ_DATA);

55          ram_d = (float*)mitrion_processor_reg_buffer(p, "ram_3", NULL, NUM_SAMPLES
                *2*sizeof(float)*8, READ_DATA);

57          return 1;
}
59
normal_arch(float *x0, float *y0, float *z0, float *k, float *l, float *m, float *
    n, float *c, float *x1, float *y1, float *z1)
61{
            int i;
63          int n;

65          // Load the 131072 generated samples 2 times to fill 262144 (2^18) blocks
                of ram.
            for(n = 0; n < 2; n++)
67          {
                    for(i = 0; i < NUM_SAMPLES; i++)
69                  {
                            int offset = i * 4;
71                          ram_a[(offset+0)+x_10*NUM_SAMPLES] = x0[i];
                            ram_a[(offset+1)+x_10*NUM_SAMPLES] = y0[i];
73                          ram_b[(offset+0)+x_10*NUM_SAMPLES] = z0[i];
                            ram_b[(offset+1)+x_10*NUM_SAMPLES] = k[i];
75                          ram_a[(offset+2)+x_10*NUM_SAMPLES] = l[i];
                            ram_a[(offset+3)+x_10*NUM_SAMPLES] = m[i];
```

```
77                      ram_b[(offset+2)+x_10*NUM_SAMPLES] = n[i];
                        ram_b[(offset+3)+x_10*NUM_SAMPLES] = c[i];
79              }
        }
81
        clock_t start_fpga, end_fpga;
83
        // Begin time measurement.
85      start_fpga = clock();

87      // Run FPGA design 2^12 times for a total of 2^12*2^18, or 2^30 total
           samples.
        for(i=0; i<4096; i++)
89      {
                mitrion_processor_run(p);
91
                mitrion_processor_wait(p);
93      }

95      // End time measurement.
        end_fpga = clock();
97
        printf("Elapsed␣CPU␣Time␣=␣%16.5lf␣seconds\n", (end_fpga-start_fpga)/(
           double)CLOCKS_PER_SEC);
99
        for(n=0; n<2; n++)
101     {
                for(i = 0; i < NUM_SAMPLES; i++)
103             {
                    union hex_float x0_h, y0_h, z0_h, k_h, l_h, m_h, n_h, c_h;
105
                        x0_h.f = x0[i];
```

```
107                     y0_h.f = y0[i];
                        z0_h.f = z0[i];
109                     k_h.f = k[i];
                        l_h.f = l[i];
111                     m_h.f = m[i];
                        n_h.f = n[i];
113                     c_h.f = c[i];

115                     // Display calculation inputs.
                        printf("%ld: x0 = %x, y0 = %x, z0 = %x, k = %x, l = %x, m
                            = %x, n = %x, c = %x \n", i, x0_h.l, y0_h.l, z0_h.l, k_h
                            .l, l_h.l, m_h.l, n_h.l, c_h.l);
117
                        int offset = i*2;
119                     x1[i] = ram_c[(offset+0)+n*NUM_SAMPLES];
                        y1[i] = ram_c[(offset+1)+n*NUM_SAMPLES];
121                     z1[i] = ram_d[(offset+0)+n*NUM_SAMPLES];

123             union hex_float x1_h, y1_h, z1_h;
                x1_h.f = x1[i];
125             y1_h.f = y1[i];
                z1_h.f = z1[i];
127
                // Display calculation outputs.
129             printf("%ld: x1 = %x, y1 = %x, z1 = %x \n", i, x1_h.l, y1_h.l,
                    z1_h.l);
                }
131     }

133     // End time measurement.
        end_fpga = clock();
135
```

```
        // Display time measurement
137     printf("Elapsed CPU Time = %16.5lf seconds\n", (end_fpga-start_fpga)/(double)
          CLOCKS_PER_SEC);

139 }

141 void gen_inputs(float *x0, float *y0, float *z0, float *k, float *l, float *m,
      float *n, float *c)
    {
143         int idx = 0;

145         // Generate reasonable values for x, y, z, k, l, m, n, and c.
            // 4 values of x, 4 values of y, 4 values of z, 4 values of k,
147         // 4 values of l, 4 values of m, 4 values of n, and 8 values of
            // c are used for a total of 4*4*4*4*4*4*4*8 or 131072 samples.
149
            int x0_c, y0_c, z0_c, k_c, l_c, m_c, n_c, c_c;
151
            for(x0_c = -1; x0_c < 2.1; x0_c++){
153         for(y0_c = -1; y0_c < 2.1; y0_c++){
            for(z0_c = -1; z0_c < 2.1; z0_c++){
155         for(k_c = -2; k_c < 1.1; k_c++){
            for(l_c = -1; l_c < 2.1; l_c++){
157         for(m_c = -1; m_c < 2.1; m_c++){
            for(n_c = -1; n_c < 2.1; n_c++){
159         for(c_c = 0; c_c < 7.1; c_c++)
                    {
161                 x0[idx] = x0_c;
                    y0[idx] = y0_c;
163                 z0[idx] = z0_c;
                    k[idx] = k_c;
165                 l[idx] = l_c;
```

```
                    m[idx] = m_c;
                    n[idx] = n_c;
                    c[idx] = c_c;
                    idx++;
                    }}}}}}}
            }
}

int main(int argc, char** argv)
{

            float x0[NUM_SAMPLES];
            float y0[NUM_SAMPLES];
            float z0[NUM_SAMPLES];
            float k[NUM_SAMPLES];
            float l[NUM_SAMPLES];
            float m[NUM_SAMPLES];
            float n[NUM_SAMPLES];
            float c[NUM_SAMPLES];

            float x1[NUM_SAMPLES];
            float y1[NUM_SAMPLES];
            float z1[NUM_SAMPLES];

            init_fpga();

            gen_inputs(x0, y0, z0, k, l, m, n, c);

            normal_arch(x0, y0, z0, k, l, m, n, c, x1, y1 ,z1);

            return 0;
}
```

Appendix D

ANSI-C Host Code of Normal-Vector Calculation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mithal.h"
#include "mithal_gen.h"
#include "float.h"

#define NUM_SAMPLES 131072

FPGA *f;
Processor *p;
float *ram_a;
float *ram_b;
float *ram_c;
float *ram_d;

union hex_float {
    float f;
```

```
19      long l;
};

21
int init_fpga()
23 {
        STATUS s;

25
        //The following code is provided by Mitrionics for host-FPGA interface.
27      // Allocate a FPGA.
        f = mitrion_fpga_allocate("");
29      if(f == NULL) {
        fprintf(stderr, "Could not allocate a FPGA\n");
31      exit(1);
        }

33
        // Create Mitrion processor.
35      p = mitrion_processor_create("top.bin.ufp");
        if(f == NULL) {
37      fprintf(stderr, "Could not create Mitrion processor\n");
        exit(1);
39      }

41      // Load the Mitrion processor onto the FPGA.
        s = mitrion_fpga_load_processor(f, p);
43      if(s != OK) {
        fprintf(stderr, "Could not load the Mitrion processor onto the FPGA\n");
45      exit(1);
        }

47
        // Associate virtual memory spaces with QDR SRAM memory addresses.
49      ram_a = (float*)mitrion_processor_reg_buffer(p, "ram_0", NULL, NUM_SAMPLES
           *2*sizeof(float)*4, WRITE_DATA);
```

```
51          ram_b = (float*)mitrion_processor_reg_buffer(p, "ram_1", NULL, NUM_SAMPLES
                *2*sizeof(float)*4, WRITE_DATA);

53          ram_c = (float*)mitrion_processor_reg_buffer(p, "ram_2", NULL, NUM_SAMPLES
                *2*sizeof(float)*4, READ_DATA);

55          ram_d = (float*)mitrion_processor_reg_buffer(p, "ram_3", NULL, NUM_SAMPLES
                *2*sizeof(float)*4, READ_DATA);

57          return 1;
}
59
normal_arch(float *x, float *y, float *u, float *c, float *f_val, float *fdx,
    float *fdy)
61 {
            int i;
63          int n;

65          // Load the 131072 generated samples 4 times to fill 528244 (2^19) blocks
                of ram.
            for(n = 0; n < 4; n++)
67          {
                    for(i = 0; i < NUM_SAMPLES; i++)
69                  {
                            int offset = i * 2;
71                          ram_a[(offset+0)+n*NUM_SAMPLES] = x[i];
                            ram_a[(offset+1)+n*NUM_SAMPLES] = y[i];
73                          ram_b[(offset+0)+n*NUM_SAMPLES] = u[i];
                            ram_b[(offset+1)+n*NUM_SAMPLES] = c[i];
75                  }
            }
```

```
77
        clock_t start_fpga, end_fpga;
79
        // Begin time measurement.
81      start_fpga = clock();
83      // Run FPGA design 2^11 times for a total of 2^11*2^19, or 2^30 total
           samples.
        for(i=0; i<2048; i++)
85      {
                mitrion_processor_run(p);
87
                mitrion_processor_wait(p);
89      }
91      // End time measurement.
        end_fpga = clock();
93
        // Display time measurement
95      printf("Elapsed CPU Time = %16.5lf seconds\n", (end_fpga-start_fpga)/(
           double)CLOCKS_PER_SEC);
97      for(n=0; n<4; n++)
        {
99              for(i = 0; i < NUM_SAMPLES; i++)
                {
101                 union hex_float x_h, y_h, u_h, c_h;
103                     x_h.f = x[i];
                        y_h.f = y[i];
105                     u_h.f = u[i];
                        c_h.f = c[i];
```

```
107
                        // Display calculation inputs.
109                     printf("%ld: x = %x, y = %x, u = %x, c = %x \n", i, x_h.l,
                            y_h.l, u_h.l, c_h.l);

111                     int offset = i*2;
                        f_val[i] = ram_c[(offset+0)+n*NUM_SAMPLES];
113                     fdx[i] = ram_c[(offset+1)+n*NUM_SAMPLES];
                        fdy[i] = ram_d[(offset+0)+n*NUM_SAMPLES];
115
                    union hex_float f_val_h, fdx_h, fdy_h;
117                 f_val_h.f = f_val[i];
                    fdx_h.f = fdx[i];
119                 fdy_h.f = fdy[i];

121                 // Display calculation outputs.
                    printf("%ld: f = %x, fdx = %x, fdy = %x \n", i, f_val_h.l, fdx_h.l
                       , fdy_h.l);
123                 }
            }
125 }

127 void get_inputs(float *x, float *y, float *u, float *c)
    {
129         int idx = 0;

131         float x_n;
            float y_n;
133         float c_n;

135         // Generate reasonable values for x, y, u, and c.  256 values of x,
            // 128 values of y, and 4 values of c are used for a total of
```

```
137         // 2^8 * 2^7 * 2^2 = 2^17 or 131072 samples.  u = (1+k)c^2 and
            // a value of 1 is assumed for k, which is reasonable.
139
    for (x_n = -2.0; x_n <= 3.11; x_n+=0.02) {
141     for (y_n = -2.0; y_n <= 3.11; y_n+=0.04) {
            for (c_n = 0.0; c_n <= .16; c_n+=0.05){
143             x[idx] = x_n;
                y[idx] = y_n;
145             u[idx] = 2*c_n*c_n;
                c[idx] = c_n;
147             idx++;
            }
149     }
    }
151
}
153
int main(int argc, char** argv)
155 {

157         float x[NUM_SAMPLES];
            float y[NUM_SAMPLES];
159         float u[NUM_SAMPLES];
            float c[NUM_SAMPLES];
161
            float f_val[NUM_SAMPLES];
163         float fdx[NUM_SAMPLES];
            float fdy[NUM_SAMPLES];
165
            init_fpga();
167
            get_inputs(x,y,u,c);
```

```
169
        normal_arch(x,y,u,c,f_val,fdx,fdy);
171
        return 0;
173 }
```

Appendix E

ANSI-C Sequential Implementation of Ray-Intersection Calculation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "float.h"

#define NUM_SAMPLES 131072

union hex_float {
     float f;
     long l;
};

void normal_arch(float *x0, float *y0, float *z0, float *k, float *l, float *m,
   float *n, float *c, float *x1, float *y1, float *z1)
{
```

```
    long i,n;

    clock_t start_cpu, end_cpu;

        // Begin time measurement.
    start_cpu = clock();

        // Run sequential program 2^13 times for a total of 2^13*2^17, or 2^30
           total samples.
    for(n=0; count<8192; n++)
    {
        for(i=0; i<NUM_SAMPLES; i++)
        {
                        float g = n[i]-c[i]*(x0[i]*l[i]+y0[i]*m[i]+(k[i]+1.0)*z0[i
                           ]*n[i]);
                        float h = c[i]*(x0[i]*x0[i]+y0[i]*y0[i]+(k[i]+1.0)*z0[i]*
                           z0[i])-2.0*z0[i];
                        float f = c[i]*(1.0+k[i]*n[i]*n[i]);
                        float u = (g*g-f*h)/(g+sqrtf(g*g-f*h));
                        x1[i] = u*l[i]+x0[i];
                        y1[i] = u*m[i]+y0[i];
                        z1[i] = u*n[i]+z0[i];
        }
    }

        // End time measurement.
        end_cpu = clock();

        // Display time measurement
        printf("Elapsed␣CPU␣Time␣=␣%16.5lf␣seconds\n", (end_cpu-start_cpu)/(double
           )CLOCKS_PER_SEC);
```

```
        for(i = 0; i < NUM_SAMPLES; i++)
        {
            union hex_float x0_h, y0_h, z0_h, k_h, l_h, m_h, n_h, c_h;

                x0_h.f = x0[i];
                y0_h.f = y0[i];
                z0_h.f = z0[i];
                k_h.f = k[i];
                l_h.f = l[i];
                m_h.f = m[i];
                n_h.f = n[i];
                c_h.f = c[i];

                // Display calculation inputs.
                printf("%ld: x0 = %x, y0 = %x, z0 = %x, k = %x, l = %x, m = %x, n
                    = %x, c = %x \n", i, x0_h.l, y0_h.l, z0_h.l, k_h.l, l_h.l, m_h.l
                    , n_h.l, c_h.l);

                union hex_float x1_h, y1_h, z1_h;
                x1_h.f = x1[i];
                y1_h.f = y1[i];
                z1_h.f = z1[i];

                // Display calculation outputs.
                printf("%ld: x1 = %x, y1 = %x, z1 = %x \n", i, x1_h.l, y1_h.l,
                    z1_h.l);

        }
}

void gen_inputs(float *x0, float *y0, float *z0, float *k, float *l, float *m,
    float *n, float *c)
```

```
{
        int idx = 0;

        // Generate reasonable values for x, y, z, k, l, m, n, and c.
        // 4 values of x, 4 values of y, 4 values of z, 4 values of k,
        // 4 values of l, 4 values of m, 4 values of n, and 8 values of
        // c are used for a total of 4*4*4*4*4*4*4*8 or 131072 samples.

        int x0_c, y0_c, z0_c, k_c, l_c, m_c, n_c, c_c;

        for(x0_c = -1; x0_c < 2.1; x0_c++){
        for(y0_c = -1; y0_c < 2.1; y0_c++){
        for(z0_c = -1; z0_c < 2.1; z0_c++){
        for(k_c = -2; k_c < 1.1; k_c++){
        for(l_c = -1; l_c < 2.1; l_c++){
        for(m_c = -1; m_c < 2.1; m_c++){
        for(n_c = -1; n_c < 2.1; n_c++){
        for(c_c = 0; c_c < 7.1; c_c++)
                {
                x0[idx] = x0_c;
                y0[idx] = y0_c;
                z0[idx] = z0_c;
                k[idx] = k_c;
                l[idx] = l_c;
                m[idx] = m_c;
                n[idx] = n_c;
                c[idx] = c_c;
                idx++;
                }}}}}}}
        }
}
```

```
int main(int argc, char** argv)
{

    float x0[NUM_SAMPLES];
    float y0[NUM_SAMPLES];
    float z0[NUM_SAMPLES];
    float k[NUM_SAMPLES];
    float l[NUM_SAMPLES];
    float m[NUM_SAMPLES];
    float n[NUM_SAMPLES];
    float c[NUM_SAMPLES];

    float x1[NUM_SAMPLES];
    float y1[NUM_SAMPLES];
    float z1[NUM_SAMPLES];

    gen_inputs(x0,y0,z0,k,l,m,n,c);

    normal_arch(x0,y0,z0,k,l,m,n,c,x1,y1,z1);

    return 0;
}
```

Appendix F

ANSI-C Sequential Implementation of Normal-Vector Calculation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "float.h"

#define NUM_SAMPLES 131072

union hex_float {
    float f;
    long l;
};

void normal_arch(float *x, float *y, float *u, float *c, float *f_val, float *fdx,
    float *fdy)
{
```

```
    long i,n;
    float v, a, p, q, r, s, w, b, dx, dy, e;

    clock_t start_cpu, end_cpu;

        // Begin time measurement.
    start_cpu = clock();

        // Run sequential program 2^13 times for a total of 2^13*2^19, or 2^30
           total samples.
    for(n=0; n<8192; n++)
    {
        for(i=0; i<NUM_SAMPLES; i++)
        {
            v=u[i]*(x[i]*x[i]+y[i]*y[i]);
            a = sqrtf(1.0f-v);
            p=1+a;
            q=a*p;
            r=p*q;
            s=2*q;
            w=c[i]/r;
            b=w*(s+v);
            dx=b*x[i];
            dy=b*y[i];
            e=sqrtf(dx*dx+dy*dy+1.0f);
            f_val[i]=1/e;
            fdx[i]=f_val[i]*dx;
            fdy[i]=f_val[i]*dy;
        }
    }

        // End time measurement.
```

```
47      end_cpu = clock();
49          // Display time measurement
        printf("Elapsed␣CPU␣Time␣=␣%16.5lf␣seconds\n", (end_cpu-start_cpu)/(double)
           CLOCKS_PER_SEC);
51
        for(i = 0; i < NUM_SAMPLES; i++)
53      {
55          union hex_float x_h, y_h, u_h, c_h;
            x_h.f = x[i];
57          y_h.f = y[i];
            u_h.f = u[i];
59          c_h.f = c[i];
61                  // Display calculation inputs.
            printf("%ld:␣x␣=␣%x,␣y␣=␣%x,␣u␣=␣%x,␣c␣=␣%x␣\n", i, x_h.l, y_h.l, u_h.l,
               c_h.l);
63
            union hex_float f_val_h, fdx_h, fdy_h;
65          f_val_h.f = f_val[i];
            fdx_h.f = fdx[i];
67          fdy_h.f = fdy[i];
69                  // Display calculation outputs.
            printf("%ld:␣f␣=␣%x,␣fdx␣=␣%x,␣fdy␣=␣%x␣\n", i, f_val_h.l, fdx_h.l, fdy_h.
               l);
71
        }
73  }
75  void gen_inputs(float *x, float *y, float *u, float *c)
```

```
{
77      long idx = 0;

79      float x_n;
        float y_n;
81      float c_n;

83          // Generate reasonable values for x, y, u, and c.  256 values of x,
            // 128 values of y, and 4 values of c are used for a total of
85          // 2^8 * 2^7 * 2^2 = 2^17 or 131072 samples.  u = (1+k)c^2 and
            // a value of 1 is assumed for k, which is reasonable.
87
        for (x_n = -2.0; x_n <= 3.11; x_n+=0.02) {
89          for (y_n = -2.0; y_n <= 3.11; y_n+=0.04) {
                for (c_n = 0.0; c_n <= .16; c_n+=0.05){
91                  x[idx] = x_n;
                    y[idx] = y_n;
93                  u[idx] = 2*c_n*c_n;
                    c[idx] = c_n;
95                  idx++;
                }
97          }
        }
99  }

101 int main(int argc, char** argv)
    {
103
        float x[NUM_SAMPLES];
105     float y[NUM_SAMPLES];
        float u[NUM_SAMPLES];
107     float c[NUM_SAMPLES];
```

```
109     float f_val[NUM_SAMPLES];
        float fdx[NUM_SAMPLES];
111     float fdy[NUM_SAMPLES];

113     gen_inputs(x,y,u,c);

115     normal_arch(x,y,u,c,f_val,fdx,fdy);

117     return 0;
}
```